(2)

COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-4055

Microsupercomputers: Design and Implementation

Stanford University
Computer Systems Laboratory

Semi-Annual Technical Report

Defense Advanced Research Projects Agency

For the period of November 1989 - March 1990

Contract Number: N00014-87-K-0828

Principal Investigator
John L. Hennessy

Associate Investigator
Mark A. Horowitz

Semi-Annual Technical Progress

November 1989 - March 1990

Contract No. N00014-87-K-0828

Order No. 1133

R & T Project Code: 4331685

Accesion For	
NTIS CRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By per call	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

Dist. "A" per telecon Dr. Andre van Tilborg. Office of Naval Research/ code 1133.
VHG 11/13/90

This work is supported by the Defense Advanced Research Projects Agency and the Office of Naval Research.

Table of Contents

Executive Summary:

Parallel Processor Architecture —

The primary focus of our architecture effort has been on completing the design of DASH and starting the construction of the prototype. The design of the prototype is now complete and frozen. We have already begun making the necessary modifications to existing SGI 4D/240 boards. The logic design of the directory board (the key component of DASH) is now complete and entered into a VALID CAE system. It is currently being simulated. The network chips are back from MOSIS. We have tested them and they are functional. In cooperation with Silicon Graphics, we have also begun work on the operating system for DASH. Our overall goal is to have the hardware for the prototype put together in the next 4 months, when we can proceed with initial hardware and software debug.

We have also been working on novel techniques to reduce the invalidation traffic and directory-memory requirements for large-scale multiprocessors. We have come up with two techniques for lowering invalidation traffic and directory memory requirements. The first technique is based on the notion of coarse vectors. It is a limited-pointer directory scheme, but uses a novel method for handling overflows. The second technique abandons the idea of having one directory entry per memory block and organizes the directory memory as a cache. This results in significant savings in the total directory memory needed. Performance results obtained so far have been very encouraging.

We have continued the development and use of our software tracing and simulation system, Tango. Tango has now been used to simulate applications running on over 100 processors. Performance evaluations have confirmed that Tango gains significant efficiency by allowing the user to specify both the information to be traced, and a memory system model with an appropriate level of abstraction.

Parallel Software —

We are building a compiler system that supports our research in a broad spectrum of topics, ranging from advanced scalar optimizations to parallel loop transformations and code scheduling for multiprocessors and superscalar processors. The synergy of the different research efforts on the same system makes possible a proper evaluation of individual compiler optimizations as well as the system as a whole. [illegible]

We have developed and began implementing a loop restructuring algorithm to improve the data locality of code for both uniprocessor and multiprocessor systems. Our approach uses a novel unified model of loop transformations.

An ongoing area of exploration is the interaction between parallel applications and architectures. Recently, an experimental study of the cache behavior of explicitly parallel code has shown that sharing is a dominant factor in multiprocessor performance. This study has led to the development of optimizations that reposition shared data to reduce sharing induced by the collocation of data in a cache block.

Work continues on investigating approaches for compiling symbolic languages (Scheme) to use parallel architectures efficiently.

3) ***Uniprocessor Architecture*** —
We have continued our work on investigating efficient methods of extracting the parallelism available at the lowest-level, by issuing multiple instruction per cycle. Our current work looks at a combined hardware-software approach, where the compiler is responsible for reordering the instructions, possibly *boosting* an instruction through a branch, and the hardware is responsible for not committing the results of these "boosted" instructions until the branch that they depend on completes. The added hardware required in this system is small, yet the performance of this system is comparable to a fully dynamically reordered machine of much greater complexity.

4) ***Computer-Aided Design*** —
Recent research has been focusing on partitioning techniques at the behavioral-level of abstraction. Partitioning is an important step in synthesis of ~~large digital~~ systems. An effective partitioning of the digital hardware can lead to simplified interconnection structure and efficient use of hardware resources. Partitioning is performed to explore area/performance tradeoffs, increase concurrency in hardware by creating multiple processes and possibly decrease complexity of control. Further, multiple pipestages can be created from functionally decomposed processes. Unlike previous approaches, partitioning at hi-level is carried out before any of the traditional synthesis tasks such as scheduling. It is believed that such an approach would lead to greater flexibility in area and performance tradeoffs than would be possible by partitioning ~~netlists~~ at the logic level. Also since there are fewer objects at ~~hi-level~~ descriptions, it is possible to do effective area/performance tradeoffs in less time.

We are investigating techniques for verifying asynchronous, concurrent designs ranging from single VLSI chips to entire systems, such as DASH. Our approach focuses on automated verification, and has already demonstrated some important successes.

Research on parallel CAD applications serves to drive our multiprocessor architecture research and to provide more powerful tools for VLSI designers. An initial system for logic simulation has been quite successful, and a new system for parallel multilevel simulation is being developed.

Our work in analyzing the power and ground noise in integrated circuits is progressing nicely. We have now used the system on a number of large CMOS and ECL circuits, and have been working on checking the errors that are introduced by various simplifications done to increase the efficiency of the program. In the process of checking these errors, we wrote a very efficient accurate resistance extractor, improving on the work done by Steve McCormick at MIT, and looked closely at modeling the effect of the substrate current. The latter is difficult, since it involves determining where the current from the bottom-plate of the wire capacitors is collected by the substrate and injected into the power-supply net. Many systems ignore this current entirely, even though it is equal to the current that flow through the driving transistor.

5) ***VLSI Design***

During this period we have continued our work in high-speed BiCMOS circuits. We have successfully used Bisim to simulate the BiCMOS sRAM, as well as completed testing the fabricated device. We have also done extensive testing on the BiCMOS Translation Look-Aside Buffer [23] and a BiCMOS PLA.

([illegible])

Technical Progress

Parallel Processor Architecture

An architectural overview of DASH appears in a paper attached to this report.

Hardware Status

During the past six months, our focus has been on completing the design and beginning the construction of the DASH prototype. This includes several sub-efforts:

(i) To be able to use the Silicon Graphics 4D/240 systems as the nodes of DASH, a number of modifications are necessary to the processor, memory, and I/O boards. For example, we need to modify the processor boards and the I/O board (which has the arbiter) to force the processor off the bus when it makes a remote-memory request and eventually be able to respond to the processor when data has been retrieved. The processor boards have also to be modified to support cache-to-cache transfer when the data is in shared state. The memory boards need to be modified to recognize non-zero base addresses plus a host of other changes. We have finalized the changes that are necessary after detailed discussions with the Silicon Graphics designers. We are currently making these changes to the boards of two SGI systems. We have also developed diagnostic suites to verify the correctness of our modifications.

(ii) The directory board forms the heart of the DASH multiprocessor. There is one such board on each SGI 4D/240 node. The board contains (a) directory memory (keeps track of caching state of blocks), (b) the directory controller (manages the directory memory), (c) pseudo-CPU (services requests from remote CPUs), (d) reply controller (handles replies to this cluster's remote requests), (e) network interface, and (f) the performance monitor. At this point the logic design of all of these subcomponents is complete and entered into our VALID CAE tools. We are currently doing simulations to check the correctness of the logic and to do timing verification. Within the next 1-2 months we expect to send the board out for fabrication.

(iii) In parallel with the work on hardware we have also built a detailed software simulator of the DASH architecture. The simulator closely follows the hardware of SGI 4D/240 and our design of the directory board. We are using this simulator to debug the distributed cache-coherence protocol used in DASH using pseudo-random test generation methods. We are also using this simulator to obtain test vectors for simulation of directory board on VALID tools. The simulator reinforces the need for experimental hardware, since it runs 10,000 times slower (per processor) than DASH.

(iv) Finally, in close cooperation with Silicon Graphics Inc., we are modifying the existing operating system on the 4D/240 to work on DASH. This includes modifying the memory management code, interrupt handlers, TLB consistency mechanisms. Much work is also being done on modifying and developing new low-level diagnostics that will check the health of the system when it first comes up and for initially configuring the hardware.

Our overall goal is to have a prototype of DASH with 16 processors to be up in about six months from now. This will involve getting all the hardware put together in about four months, leaving two months for initial hardware and software debug.

Basic Architecture Studies and Simulation Tools Effort
In addition to the work on the prototype, we have been working on novel techniques to reduce the invalidation traffic and directory-memory requirements for large-scale multiprocessors. Recall that directory-based cache-coherence schemes rely on a directory to keep track of all processors caching a memory block. When a write to that block occurs, point-to-point invalidation messages are sent to keep the caches coherent. A straightforward way of recording the identities of processors caching a memory block is to use a bit vector per memory block, with one bit per processor. Unfortunately, when the main memory grows linearly with the number of processors, the total size of the directory memory grows as the square of the number of processors, which is prohibitive for large machines. To remedy this problem several schemes that use a limited number of pointers per directory entry have been suggested [1]. Unfortunately, these schemes often cause excessive invalidation traffic.

We have proposed two methods for lowering invalidation traffic and directory memory requirements. The first is the *coarse vector* directory scheme. In the most common case of a block being shared between a small number of processors, the directory is kept in the form of several pointers. Each points to a processor which has a cached copy. When the number of processors sharing a block exceeds the number of pointers available, the directory switches to a different representation. The same memory that was used to store the pointers is now treated as a coarse bit-vector, where each bit of the state indicates a group of processors. We term this new directory scheme Dir_iCV_r, where i is the number of pointers and r is the size of the region that each bit in the coarse vector represents. With all bits set, the equivalent of a broadcast is achieved. While using the same amount of memory, the proposed scheme is at least as good as the limited pointer scheme with broadcast -- presented as Dir_iB in [1].

The second method reduces directory memory requirements by organizing the directory as a cache, instead of having one directory entry per memory block. Since the total size of main memory in machines is much larger than that of all cache memory, at any given time most memory blocks are not cached by any processor and the corresponding directory entries are empty. The idea of a directory cache that only contains the active entries is thus appealing. Furthermore, there is no need to have a backing store for the directory cache. The state of a block can safely be discarded after invalidation messages have been sent to all processor caches with a copy of that block. Our scheme of directory caches brings down the storage requirements of main memory based directories closer to that of cache-based linked list directory schemes such as the SCI scheme. However, we avoid the disadvantages of cache-based directories -- having to follow linked lists to determine the processors caching a memory block and having to build the directory out of expensive SRAM cache memory.

We have done limited comparisons of the full bit-vector scheme and existing limited pointer schemes with our coarse vector scheme. We have also completed preliminary evaluations of cached directories. The performance results obtained so far have been encouraging. (We used the data and invalidation traffic generated by the different schemes as a measure of performance.) Our results show that depending on the application, the coarse vector scheme produces up to 65% less memory message traffic than the next best limited pointer scheme. Caching the directory adds less than 10% to the traffic while potentially reducing directory memory overhead by an order of magnitude.

We have continued the development and use of our software tracing and simulation system, Tango. Tango has now been used to simulate applications running on over 100 processors. Performance evaluations have confirmed that Tango gains significant efficiency by allowing the user to specify both the information to be traced, and a memory system model with an appropriate level of abstraction. Tango has been extended to run with applications written in Fortran as well as C, which increases the set of applications available to us for study. Tango is currently being used to support a variety of investigations in algorithms for logic simulation and linear algebra, shared data locality, program synchronization characteristics, memory system design, and the DASH architecture.

Our initial DASH studies used a detailed cycle-by-cycle simulation of the memory system, and we found that most of the time in Tango was spent simulating the memory system. This discovery prompted efforts to consider faster approaches to DASH memory system simulation. We now have several alternative simulators. One is an event-driven simulator that is 2-5 times faster. We also have simulators that approximate contention and cache behavior in DASH; these simulators allow the user to trade-off modest amounts of accuracy for simulations that are orders of magnitude faster, and are especially valuable when studying the behavior of applications running on DASH.

We have recently begun the design of interactive visualization tools that will aid users in understanding simulation results. An interface to Tango will permit displaying both general information about application behavior, and also information that is specific to the memory system used in the simulation.

Parallel Software

Overview of the Compiler Development

We have developed a solid platform for research in scalar and parallel code compilation and compiler/architecture tradeoffs. We have implemented a basic system that compiles Fortran code into MIPS assembly code. Integrating the information used in scalar and parallel code optimizations into one representation, this system is supportive of research into a wide range of topics. Our current investigation includes the following key compiler issues:

- A scalar data flow optimizer generator
- Measurement on the effectiveness of data dependence tests
- Measurement of parallelism in applications
- A new technique to analyze the data flow of array data
- Loop level transformations for parallelism and data locality
- Superscalar code scheduling
- Data placement for multiprocessors

Developing a compiler system for uni- and multiprocessors can potentially entail a significant laborious implementation effort. Fortunately, we have been able to develop new insights and algorithms for many of the major components of the system and increase the research value of our implementation effort. For example, even in the well studied domain of scalar code optimizations, we have developed the new concept of a data flow optimizer generator and we are in the process of implementing the algorithm. Similarly, as we study the problem of data locality, we have developed a unifying approach to general loop transformations and greatly simplified the implementation of such optimizations.

Through the broad coverage of this research project, we hope to develop a complete, fully optimizing compiler. In this way, we can evaluate individual optimizations properly, within the context of all other optimizations. More importantly, this approach allows us to evaluate optimizing compilers as a whole and identify the true strengths and weaknesses of the compiler technology and how it must advance to better support parallel processing. A recent paper covering some of this work is attached this report.

Integrating Data Flow with Data Dependence

The integration of scalar and parallel code optimizations in one system fosters the development of new fundamental compiler techniques. The current state of the art is that data flow analysis does not distinguish between different elements of an array, whereas data dependence analysis disambiguates array references but is insensitive to data flow. If data flow analysis can be combined with data dependence information, the new analysis data can give rise to new code optimizations such as allocating array references in registers and expanding array variables to generate more parallelism in multiprocessor programs.

Unlike previous approaches that augment data flow techniques to provide more precise information on arrays, we are investigating the technique of extending the data dependence technique to account for data flow. Stimulated by the motivation to support new optimizations, we have developed a new data dependence representation that appears to better support even the existing code transformations.

The key insight is to characterize each array reference by its position on an absolute scale. Any relationship between any two or more references can be derived simply and directly from this representation. This is in contrast to the existing approach where the dependence relationships between every pair of accesses are explicitly represented. It is costly to generate the pair-wise information in the first place, and it is even more so to derive information relating more than two cells from this pair-wise representation. Moreover, while it is trivial to update our dependence information after each transformation, existing systems tend to completely re-evaluate all dependences after each code optimization phase.

In this initial study, we concentrated only on array references that sweep the entire array space exactly once in the entire loop nest. As the results we obtain are quite promising, we plan to extend this model to more general references, and to formulate the legality of existing loop level transformations using this new representation.

Optimizations for Data Locality and Parallelism

While the computation bandwidth of processors experiences phenomenal increases in recent years, the main memory subsystem fails to keep pace and thus the gap between processor and memory speeds continues to widen. It is no longer feasible to ignore the memory hierarchy; compilers must improve the cache performance by increasing the locality of data. We have developed and implemented a compiler optimization algorithm that restructures numerical kernels to maximize data locality. Besides finding data locality, our algorithm can also discover both fine and coarse grain parallelism, the former exploitable on a superscalar processor and the latter on a multiprocessor. Our current implementation generates runnable code for sequential machines. As for multiprocessor code, the implementation can already identify the parallelism, and development of a backend to generate runnable multiprocessor code is in progress. Our next step is to experiment with the implementation using benchmark programs.

In focusing on this new problem of data locality, we have developed techniques that are important to the general field of loop optimizations. Numerous loop transformations (e.g. loop interchange, reversal, skewing and tiling) have been proposed and shown to be useful for vectorization and concurrentization. An issue critical to using these optimizations in practice is how to decide when and which of these transformations should be applied.

Each individual transformation had a different legality test. However, individual transformations may not all directly contribute to improving a particular goal, but may nonetheless be important for making other optimizations legal. This has led to the proposed "generate and test" technique which generates and evaluates all programs obtained through all different possible permutations of transformations.

Our approach is to unify all legality tests of transformations and their effects in the same mathematical model of iteration space and dependence vectors, so we can define the legality test and effect of *compound* transformations. Under this model, a compound transformation on sequential loops is legal simply if the dependence vectors after transformation remain lexicographically positive. By expressing the goal of the transformations as a desired effect on the code, we can obtain the best compound transformation efficiently.

Through the analysis under this unified framework, we show that tiling is the key transformation useful for generating data locality and parallelism. The problem of data locality is reduced to placing the maximum number of loops identified to carry locality in the innermost tile. Using this goal and the legality considerations of tiling, we can significantly prune the search space to find the best compound transformation. The conditions that made tiles legal in the first place guarantee both coarse and fine grain parallelism within a tiled loop. Thus, we have an extremely simple algorithm that can generate programs that contain both parallelism and locality.

This unification and simplification of the conceptual model is particularly significant when it comes to implementation. In the previous approach where only the effects of individual transformations are understood, a compound transformation has to be implemented through a series of transformations. With each transformation, the code (e.g. the loop bounds) becomes more complex due to the effects on the boundary conditions. However, if the compound transformation is performed on the original simple source program, the resulting code and thus the implementation are much simplified.

Improving the Cache Performance of Explicitly Parallelized Code
Another component of our research in cache optimizations focuses on explicitly parallelized applications for multiprocessors that support cache coherence in hardware. In this case, it is the behavior of the shared data that dominates the cache performance, which in turn is a critical factor in determining the multiprocessor performance. We have performed some exploratory research to understand and quantify the significance of sharing in application benchmarks. In addition to investigating the effect of the compiler and its role in improving the cache performance, we also study the implication of sharing on the cache design.

Most studies of the behavior of shared data have been based on unoptimized code. By measuring code compiled with an optimizing compiler, we found a much higher fraction of shared references in the data stream than was previously suggested. Traditional compiler optimizations optimize away non-shared memory references and thus substantially increase the significance of shared data references. The fraction of shared references in the data stream more than doubled on average, reaching values from 33% to over 90%. Since the programs run faster after optimization while the number of shared references remains the same, the shared data access bottleneck is intensified. The large discrepancy in shared reference ratios between optimized and unoptimized code suggests that performance studies of multiprocessor programs must be based on optimized code for the results to be meaningful.

The cache miss rate and the traffic created by sharing depend on two effects: *true sharing*, which is intrinsic to a reference stream of a parallel program, and *false sharing*, which is induced by the collocation of different data items in the same cache block. Measurements on parallel programs indicate that false sharing misses generally increase with block size, while cold and true sharing misses generally decrease with block size. However, the true sharing miss rate drops more slowly than might be expected, leading in some cases to an increase in the overall miss rate and in almost all cases a dramatic growth in the amount of shared data traffic with increasing block sizes. Most of this traffic is composed of words that will not be referenced by a given processor between two consecutive misses on the block by the same processor. This suggests that short data cache blocks should be used to maximize performance.

The analysis suggests two approaches to improve the cache behavior of the programs: repositioning the shared data in memory at the block level, and changes in the data structure organization or in the assignment of computation to processors. The first approach is appropriate for applications with substantial false sharing misses; the second one is especially appropriate for programs with low false sharing and a very slow decrease in the cold and true sharing misses with block size increases. The first approach is evaluated and demonstrated to have a significant impact on some of the applications studied. We expect that these optimizations will be even more important in future large-scale multiprocessors. Firstly, higher communication latencies are likely to be involved, increasing the penalty of each cache miss. Secondly, with larger numbers of processors, the ratio of misses eliminated over program execution time will possibly be higher.

Even after the placement optimizations, we observe that multiprocessor applications may have significantly high miss rates than the uniprocessor miss rates. For large multiprocessors to be effective on the type of programs considered, advances need to be made in reducing the miss rates by better algorithm design, and in building hardware and software systems that tolerate the high cache miss rate and traffic patterns of shared data. We are currently in the process of studying the feasibility of program-directed cache prefetching.

Scheme

This research focuses on automatic program transformations for efficiently implementing scientific and symbolic programs on serial and parallel machines. After implementing several toy systems to flesh out our ideas and techniques, we are ready to implement a fully automatic partial evaluator that handles all of the Scheme programming language. We have shown that partial evaluation exposes tremendous amounts of instruction level parallelism. This is an important feature for harnessing the power of superscalar and superpipelined architectures. To date our results have been obtained using a programmer directed partial evaluator. We feel that our techniques and results will become fully acceptable only when they become fully automatic. (A separate proposal has been submitted in response to BAA#90-03 to fund the construction of this partial evaluator.)

On a separate topic, Katz and Weise have invented methods for making the future construct of MultiLisp and MultiScheme interact properly with the continuation mechanism of Scheme. Previous implementations exhibit pathological behavior when futures and continuation were used together.

Uniprocessor Architecture

Boosting Beyond Static Scheduling in a Superscalar Processor

We continue to search for opportunities where new combinations of software and hardware functionality can propel the performance level of processors to a new height. An important problem that demands creative cooperation between software and hardware is the parallelization of non-numerical code.

Software techniques, e.g. software pipelining, have previously been shown to be effective in exploiting the parallelism in statically-scheduled superscalar or VLIW (Very Long Instruction Word) machines. This approach relies on the numerous "do-loop" branches that can be evaluated early to expose the parallelism across many iterations. However, non-numeric code has more data dependent branches that cannot be resolved early. Hardware-scheduled superscalar machines can parallelize nonnumeric code better by speculatively executing operations beyond a branch before the branch decision is resolved. Unfortunately, the additional hardware necessary to look far ahead in the dynamic instruction stream, find independent operations, and schedule these independent operations out of order is costly and complex. The complexity may lengthen the cycle time of the machine and reduce the actual speedup over the scalar processor.

We have developed an architecture that combines the best qualities of static and dynamic instruction scheduling to increase the performance of non-numerical applications [20]. The architecture performs all instruction scheduling statically to take advantage of the compiler's ability to efficiently schedule operations across many basic blocks and the lower hardware cost. Since the conditional branches in non-numerical code are highly data dependent, the architecture introduces the concept of *boosted* instructions, instructions that are committed conditionally upon the result of later branch instructions. Boosting effectively removes control dependence as a scheduling constraint and makes the scheduling of side-effect instructions as simple as those that are side-effect free. For efficiency, boosting is supported in the hardware by *shadow structures* that temporarily hold the side effects of boosted instructions until the conditional branches that the boosted instructions depend upon are executed. When the branch condition is determined, the buffered side effects are either committed or squashed.

Our experiments on non-numerical code show that adding a single level of boosting to a statically-scheduled superscalar processor yields a 1.6-times speedup over scalar code. This performance is comparable to the performance of an aggressive, dynamically-scheduled superscalar processor. Consequently, we are working on a compiler and a simulator for the full functionality of boosting. This system will experiment with the boosting of instructions above multiple conditional branches, the reorganizing of instructions both up and down across multiple basic block boundaries, and the advantages of memory disambiguation.

Even if hardware scheduled superscalars could be made to run at the same clock speed, architectures with boosting are still faster. Sophisticated compilers can identify the operations on the critical path. By boosting these operations up branches, the compiler can cause these operations to execute even before they could have been fetched were the machine a hardware-scheduled superscalar. As we scale up the degree of parallelism in the machine, this advantage will become even more significant due to a higher ratio of branches per instruction.

Computer Aided Design

Synthesis

In our approach the hardware behavior is first abstracted in terms of a sequencing graph model. The problem of hardware partitioning is then formulated as a graph partitioning problem. The input graph model may be hierarchical with externally imposed resource and timing constraints. Area and latency estimates for hierarchical graphs are computed in a bottom-up manner. Once an optimum partition is determined, a set of behavior-preserving transformations are applied to the sequencing graphs in order to obtain two interacting sequencing graphs.

We have applied Kernighan-Lin heuristics to hardware partitioning with the main objective of performing area-directed cuts, such that a cost function of overall latency and intercommunication cost is minimized. In addition constraints are placed on the size and pin-out of the individual blocks. Presently we are investigating: ways of performing incremental scheduling during the partitioning process, strategies to handle timing constraints between sequencing graphs, partitioning in presence of unbounded delay operations, and resource sharing between partitioned blocks.

Formal Verification of Concurrent Hardware
Multiprocessor hardware is highly concurrent, by its nature. Unfortunately, concurrent hardware designs are difficult to debug by the usual methods of simulation and prototyping, because concurrent systems behave nondeterministically. This nondeterminism leads to transient and nonrepeatable problems that are difficult to localize and diagnose.

One potential solution to this problem is formal verification using automated theorem-provers. Formal hardware verification has been improving rapidly, to the point where simple processors have been verified. However, existing techniques are not necessarily effective for concurrent systems. For example, architectural features that depend on concurrent interaction have been omitted from almost all formally verified processor designs. These include caches, pipelining, interrupts, and page mapping. We have addressed this deficiency by developing techniques for using existing theorem-provers for concurrent designs. Recently, we have had some success in using the HOL ("Higher-Order Logic") system to verify a simple sequential cache and a more sophisticated directory-based multiprocessor cache.

In our verification method, the implementation (system behavior) and specification (desired behavior) are both represented as state graphs. The graphs can have an infinite number of states, since the states are defined by logical formulas, not represented explicitly. To prove that the implementation meets a specification, one gives a logical definition of a correspondence between the states of the two graphs. This correspondence is called a *refinement relation.* The properties of a refinement relations basically ensure that for every execution path through the implementation, there is a parallel path through the specification (indicating that the implementation path is desirable). The theorem-prover is used to show that the supposed refinement relation really IS a refinement relation.

This technique is sufficient for proving *safety properties*, which assert (intuitively) that "nothing bad happens". For example, a processor receiving an inconsistent value from the cache because of a protocol bug would be a violation of a safety property. Techniques for proving liveness properties (e.g. that every read request is eventually satisfied) have yet to be developed.

The proof of the simple cache is complete; the proof of the multiprocessor cache is 80% complete. Papers describing this work have been submitted to the International Conference on Computer Design and to the Workshop on Computer-Aided Verification (to be held in June, 1990).

Power and Ground Noise Analysis
As circuit technologies scale, the resistance of the wires on the circuit become larger relative to the effective resistance of the scaled transistors. This problem is made worse by the even increasing die size. The combination of these effects makes the problem of iR drop in the power distribution wiring

an increasingly significant problem for the VLSI designer, and a problem that is likely to get worse as technologies improve. To better understand the problem, and to provide some means for a designer to check the supply drops in his/her chip, we have written a software system called Ariel that can determine the supply drops in a VLSI chip. The system works for both CMOS and ECL circuits, and has been used to check a number of large chips.

The system consists of three main components, a resistance extractor to create a resistor network model of the power supply line, a switch level simulator that captures when nodes transition and the shape of the current pulse that is generated, and a linear solver for sparse equations that is used to generate the node voltages from the injected currents and the power supply model. Since the systems we are interested in solving have large power supply nets with 10-100 thousand transistors connected to them, efficiently in the algorithms is extremely important. We have been able to run the complete system on a number of CMOS CPU chips.

Our recent work has been focused on checking some of the approximations Ariel makes to increase efficiency. There are two main places where approximations are used. The first is in the extraction of the power-supply resistance network. Rather than using an exact solution, a polygonal approximation is used, which greatly reduces both the number of nodes in the resulting network, and the time required to generate the network. We have recently finished writing an accurate, efficient resistor extractor that is based on the work done by Steve McCormick at MIT, but uses a more efficient solver for Laplace's Equation. Although this system is much slower than the original extractor, it does allow use to compare the results of the two systems. These results indicate that the voltage found by the simpler extractor are within 20% of the voltages using the true resistance (worse case in the examples we have tried), and the nominal results are very close.

The other approximation is in modeling how the substrate current enters the power-supply network. Since the currents through the two terminal of a capacitor are equal the current from the bottom-plate of the capacitor is equal to the current that flows from the top-plate through the transistor. Ariel uses a simple bounding box model, it takes the bounding box of the wire being driven, and divides the substrate current evenly between all the substrate contacts contain in the bounding box. To check this approximation we have changed the system to more carefully divide up the substrate under the wire and distribute the current more accurately. The difference in the calculated voltages for this change are small, on the order of 5% or less.

We plan to continue our work on this system to bring it to the point where others can use the tool. We have already had interest from a number of chip design companies to get a copy of the system.

Simulation

Parallel Simulation

CAD applications have been proposed to further explore the potentials of the DASH multiprocessor machine and parallel paradigms in general. Research has been done on integration of existing different simulators to obtain a parallel multi-level simulator. Parallelism is obtained at each simulator level by decomposing its simulation into smaller blocks and managing the communication and synchronization of these blocks.

Initially, algorithms for distributed and parallel simulation were investigated. The framework of interest was a distributed discrete-event simulation within a set of communicating elements that exchange information through messages and distributed time. We started with the well-known time-stamped algorithms (Chandy and Misra) to reduce deadlock occurrence, increase concurrency, and reduce communication traffic overhead. The concepts of intervals, rather than time stamps, are used to represent the period of time during which an event is valid in a simulation. With their use there are situations in which a simulated element will not block while it would have otherwise if time-stamped messages were used. With the use of this kind of messages, it is proved that acyclic network simulations are deadlock free. The use of labels on interval messages was introduced to allow messages to detect loops and obtain event scheduling optimizations. An interesting feature of intervals is that it allows simulation of future intervals in present time. These results were presented in the SCS Multiconference on Modeling and Simulation on Microcomputers [24].

A prototype to test and explore the concept of parallel mixed-mode simulation is under development. The simulators being integrated consist of THOR, a behavioral simulator for use with digital circuits at either the functional, register transfer or gate level, IRSIM, a switch-level simulator for MOS circuits, and SPICE, a general-purpose circuit-level simulator. This prototype is like a printed circuit board backplane where existing simulator programs are plugged in to run in parallel either at the same design level or at different design levels to function as a harmonic mixed level simulator. Basically it consist of a kernel to handle the interface to simulators and coordinate their computation. It uses the concepts mentioned above of interval messages, and labeled messages within loops. Also it handles conversion between design levels as in the case of logic signals and circuit signals.

Mixed-mode simulation in general does not impose implementation difficulties; it is the efficient parallel, or distributed, processing of such simulation that introduces design complexities. First, mixed-mode simulation requires the integration of all the functionalities that exist at each of the simulation levels. Furthermore, other levels or different scheduling algorithms are expected to appear or evolve in the future. Our solution to integrating existing programs deals with this particular problem. Second, the decomposition of the system under simulation is an important task that will affect performance. The third issue is the distributed algorithm used. Conventional Chandy-Misra algorithms -- deadlock avoidance, deadlock recovery -- suffer from performance degradation. Optimistic algorithms as used in the Time-Warp are not applicable to mixed-mode simulation because of the complexity involved in simulation roll-back. The use of time intervals and labeled messages on the actual prototype will be measured and analyzed regarding their impacts on performance. Finally, a consistent and uniform user interface capable of handling all the levels will be provided.

This prototype system is to be used for the verification of an adaptive signal processing system at both the chip and the board levels, and for the design of a complex CMOS Viterbi detection chip.

VLSI Design

We had designed a 64K BiCMOS sRAM that was fabricated in TI's 0.8u BiCMOS technology, but unfortunately the RAM contained a small number of design errors that made testing difficult. We were able to get around most of the problems by changing some reference levels and using a ion milling system, but this greatly limited the number of parts we were able to test. The resulting access time was about 3.6ns, but the parts we were able to measure were from the slow split, so we think a correct version of this RAM would be even faster. The write timing was also measured, and the minimum write pulse width was 4ns. Again since one of the errors in the design involved the write reference level, we feel that a corrected design will have an even shorter write pulse.

Since the entire design was completed in two months, we are now looking at the design in more detail, trying to find ways to either improve its performance or noise immunity, or to decrease its power dissipation. These changes would then be incorporated into a revised version of this part.

The work on the TLB was extremely successful, with the final data indicating a complete translation time of 3.6ns. Removing the input and output delay gives an internal translation time of under 3ns. We are now looking at ways to improve the TLB by trying to reduce the 600mW power it requires. It looks possible to replace some of the static pulldown currents with clocked devices without changing the access time. This basic CAM/RAM cell can be used to build a high-speed programmable logic element, and we are starting to look into this application a little more.

During this period we have also been working on Bisim, our BiCMOS simulator. We have developed a set of piece-wise linear models for bipolar and MOS devices, and have a set or algorithms for finding final voltages and timing for these piece-wise linear elements. The final-value code looks good. We were able to simulate a large class of circuit forms using it, including the BiCMOS sRAM. We are now concentrating on the timing models for these circuits. The initial model that we developed works well for networks with "gentle" non-linearities, but has difficulties solving networks with diodes, which generate sharp changes in the current flow.

Publications, Presentations, Reports

1. Agarwal, A., Simoni, R., Hennessy, J. and Horowitz, M., Scalable Directory Schemes for Cache Consistency, IEEE, *15th International Symposium on Computer Architecture*, Honolulu, HI. June, 1988.
2. Berlin, A. and Weise, D. Compiling Scientific Code using Partial Evaluation. *Computer*. 1990. Submitted for publication.
3. Dill, D. and Loewenstein, P., Verification of Multiprocessor Cache Protocol using Refinement Relations and Higher-Order Logic, Rutgers University, *Workshop on Computer-Aided Verification*, 1990.
4. Dill, D. and Loewenstein, P., Formal Verification of Cache Systems using Refinement Relations, IEEE, *International Conference on Computer Design*, 1990.
5. Gasbarro, J. and Horowitz, M., Testarossa: A Single-Chip, Functional Tester for VLSI Circuits, IEEE, *International Solid-State Circuits Conference*, San Francisco, CA. February, 1990.
6. Gharachorloo, K., Lenoski, D., Laudon, J., Gupta, A. and Hennessy, J., Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, IEEE/ACM, *17th Annual International Conference on Computer Architecture*, Seattle, WA. May, 1990. Also appears as Stanford University technical report number CSL-89-405 published December 1989.
7. Goldschmidt, S. and Davis, H., *Tango Introduction and Tutorial*, Stanford University, Computer Systems Laboratory, Technical Report, CSL-90-410, January, 1990.
8. Gupta, A. and Weber, W.-D., *Reducing Memory Traffic Requirements for Scalable Director-Based Cache Coherence Schemes*, Stanford University, Computer Systems Laboratory, Technical Report, CSL-90-417, March, 1990. Submitted for publication to International Conference on Parallel Processing.
9. Hayes, B., Anonymous One-Time Signatures and Flexible Untraceable Electronic Cash, *AUSCRYPT*, Sydney, Australia. August, 1990.
10. Horowitz, M., Slamowitz, M., Rose, B. and Johnson, M., A 3.5ns, 1 Watt, ECL Register File, IEEE, *International Solid-State Circuits Conference*, San Francisco, CA. February, 1990.
11. Katz, R. and Hennessy, J. L. High Performance Microprocessor Architectures. *Journal of High Speed Electronics*. 1, (1): February, 1990.

12. Katz, M. and Weise, D., Continuing into the Future: On the Interaction of Futures and First-Class Continuations, *ACM Conference on Lisp and Functional Programming*, 1990.

13. Lam, M., Compiler Optimizations for Superscalar Computers, *9th International Conference on Computing Methods in Applied Sciences and Engineering*, Paris, France. January, 1990.

14. Lenoski, D., Gharachorloo, K., Laudon, J., Gupta, A., Hennessy, J., Horowitz, M. and Lam, M., *Design of the Stanford DASH Multiprocessor*, Stanford University, Computer Systems Laboratory, Technical Report, CSL-89-403, December, 1989.

15. Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A. and Hennessy, J., The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor, IEEE/ACM, *17th Annual International Conference on Computer Architecture*, Seattle, WA. May, 1990. Also appears as Stanford University technical report number CSL-89-404 published December 1989.

16. Lenoski, D., Gharacharloo, K., Laudon, J., Gupta, A., Hennessy, J., Horowitz, M. and Lam, M., Design of Scalable Shared-Memory Multiprocessors: The DASH Approach, ACM, *Compcon*, February, 1990. Also appears as Stanford University technical report number CSL-89-403 published December 1989.

17. Rothberg, E. and Gupta, A., *A Comparative Evaluation of Nodal and Supernodal Parallel Sparse Matrix Factorization: Detailed Simulation Results*, Stanford University, Computer Systems Laboratory, Technical Report, CSL-90-416, February, 1990. Also appears as STAN-CS-90-1305 published under the auspices of the Computer Science Department.

18. Ruf, E. and Weise, D. LogScheme: Integrating Logic Programming into Scheme. 1990. Accepted for publication.

19. Schnorf, P. and Ganapathi, M., *Compilation of Single Assignment Languages: Analysis and Propositions*, Stanford University, Technical Report, CSL-89-399, November, 1989.

20. Smith, M., Lam, M. and Horowitz, M., Boosting Beyond Static Scheduling in a Superscalar Processor, IEEE/ACM, *17th Annual International Conference on Computer Architecture*, Seattle, WA. May, 1990.

21. Soule, L. and Gupta, A. Parallel Distributed-Time Logic Simulation. *Design & Test of Computers*. 32-48, December, 1989.

22. Stark, D. and Horowitz, M. Techniques for Calculating Currents and Voltages in VLSI Power Supply Networks. *IEEE Transactions on Computer-Aided Design*. 9, (2): 126-132, February, 1990.

23. Tamura, L., Yang, T.-S., Wingard, D., Horowitz, M. and Wooley, B., A 4-ns BiCMOS Translation Lookaside Buffer, IEEE, *International Solid-State Circuits Conference*, San Francisco, CA. February, 1990.

24. Todesco, A. and Meng, T., Interval Methods for Distributed Simulation Systems, *Proceedings on SCS Multiconference on Modeling and Simulation on Microcomputers*, San Diego, CA. January, 1990.

25. Torrellas, J., Hennessy, J. and Weil, T., Analysis of Critical Architectural and Program Parameters in a Hierarchical Shared-Memory Multiprocessor, ACM, *Sigmetrics*, May, 1990.

26. Torrellas, J. and Hennessy, J., *Estimating the Performance Advantages of Relaxing Consistency in a Shared Memory Multiprocessor*, Stanford University, Computer Systems Laboratory, Technical Report, CSL-90-365, February, 1990.

27. Tucker, A. and Gupta, A., Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors, ACM, *12th Symposium on Operating Systems Principles*, Lichtfield Park, AZ. December, 1989.

28. Weise, D. Formal Verification of MOS Circuits. *IEEE Transactions on Computer-Aided Design*. 1990. Accepted for publication.

29. Weise, D., Graphs as an Intermediate Representation for Partial Evaluation, *AAAI-90*, 1990.

Design of the Stanford DASH Multiprocessor

Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, John Hennessy, Mark Horowitz and Monica Lam

Technical Report No. CSL-TR-89-403

December 1989

This research has been supported by DARPA contract N00014-87-K-0828; Daniel Lenoski has been supported by Tandem Computers Incorporated.

Design of the Stanford DASH Multiprocessor

Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta,
John Hennessy, Mark Horowitz and Monica Lam

Technical Report: CSL-TR-89-403

December 1989

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

The Stanford DASH multiprocessor advances the state of parallel computing by combining the programmability of shared-memory machines with the scalability of message-passing machines. The key idea on which DASH is built is that of distributed directory-based cache coherence. Shared memory in the machine is distributed among the processing nodes, and processor caches are kept coherent by sending point-to-point messages between the nodes on an interconnection network. The network provides the scalable interprocessor communication bandwidth and the hardware-supported coherent caches provide for low latency access to shared data and ease of programming.

The prototype DASH machine will consist of 64 high performance microprocessors, with an aggregate performance of over 1200 MIPS and 250 scalar MFLOPS. This paper discusses the mechanisms for providing scalable memory bandwidth, reducing and tolerating memory latency, and supporting efficient synchronization. The paper also describes the preliminary design of the machine, discusses various implementation issues, and contrasts our approach to other proposals with similar goals.

Key Words and Phrases: scalable shared-memory multiprocessors, distributed directory-based cache coherence, DASH, memory latency, memory consistency, synchronization.

Design of the Stanford DASH Multiprocessor

Daniel Lenoski, James Laudon, Kourosh Gharachorloo,
Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

The Stanford DASH multiprocessor advances the state of parallel computing by combining the programmability of shared-memory machines with the scalability of message-passing machines. The key idea on which DASH is built is that of distributed directory-based cache coherence. Shared memory in the machine is distributed among the processing nodes, and processor caches are kept coherent by sending point-to-point messages between the nodes on an interconnection network. The network provides the scalable interprocessor communication bandwidth and the hardware-supported coherent caches provide for low latency access to shared data and ease of programming.

The prototype DASH machine will consist of 64 high performance microprocessors, with an aggregate performance of over 1200 MIPS and 250 scalar MFLOPS. This paper discusses the mechanisms for providing scalable memory bandwidth, reducing and tolerating memory latency, and supporting efficient synchronization. The paper also describes the preliminary design of the machine, discusses various implementation issues, and contrasts our approach to other proposals with similar goals.

1 Introduction

We are currently building a shared-memory multiprocessor, called DASH (Directory Architecture for SHared-memory), at Stanford's Computer Systems Laboratory. The fundamental premise of this research is that it is possible to build a *scalable* shared-memory machine by using a distributed directory-based cache coherence protocol. The shared memory in our machine is partitioned among the processing nodes, which are connected by a scalable interconnection network. A distributed directory associated with main memory stores the identities of all processors caching each memory block. This allows processor caches to be kept coherent by sending point-to-point messages. The DASH architecture combines the advantages of several existing and proposed architectures into a single system. These advantages include support for shared memory to make programming easy, coherent caches to reduce memory latency and make the system efficient, and a directory-based coherence protocol to provide scalability.

DASH is designed to be a general purpose machine. Our goal is to support a wide variety of applications in engineering and science—not be restricted to a few highly structured parallel applications.

Therefore, it is important to support a shared-memory model, where processors can directly access all memory in the system. Direct access eases the problem of data partitioning and dynamic load distribution. More importantly, shared memory makes it easier to implement parallelizing compilers, which we believe are essential for parallel processing to gain widespread usage.

Current shared-memory machines have one of two drawbacks: either they are not scalable, or if they scale well, they do not support coherent caches, which significantly reduces the performance of the individual processors.

Examples of non-scalable multiprocessors are the Encore Multimax and Sequent Symmetry. The primary reason for their non-scalability is their bus-based interconnect. While the bus provides a fixed bandwidth interconnect, the bandwidth requirements of these machines increase linearly with the number of processors. There are two reasons for this. First, the total bandwidth needed to service regular cache misses increases linearly with the number of processors. Second, the bandwidth required for invalidation and update messages to keep the caches coherent increases linearly, since the fraction of memory references per processor that cause coherence traffic is expected to stay constant as processors are added. On the positive side, bus-based machines are easy to design, and are effective when only a small number of processors is needed.

The BBN Butterfly [20] and IBM RP3 [18] multiprocessors are examples of scalable machines. Both these machines distribute the shared memory among the processors and use a scalable interconnection network. However, existing machines in this class either do not provide caches (as in Butterfly), or do not provide for hardware cache coherence (as in RP3). In the latter case they either don't cache shared writable data, or rely on compiler or programmer directed cache management. Unfortunately, compiler technology is not advanced enough to do a good job of deciding what to cache and for how long. For some classes of applications and languages (e.g., C-based programs), it is unclear that compiler technology will ever be effective. When high performance processors are used, the performance of this class of machines is primarily limited by the shared-memory accesses that have to go over the network.

In contrast to these machines, DASH provides for both scalability and hardware coherent caches. The high-level organization of the machine is shown in Figure 1. The architecture consists of a number of processing nodes connected through a low-latency interconnection network. Physical memory is distributed among the nodes. Each processing node, or *cluster*, consists of a small number of high-performance processors with their individual caches and a portion of the shared memory connected by a snooping bus. The cluster interfaces to the interconnection network through the directory controller. The directory controller includes directory memory that tracks the remote caching state of the memory within the cluster, and the *remote access cache* which tracks the outstanding requests made by the processors within the cluster. The remote access cache also provides a buffer for incoming network replies. Its snoopy cache structure allows it to merge outstanding requests from different processors and supplement the functionality of the processor caches. The snoopy scheme keeps caches coherent within a cluster while inter-cluster coherence is maintained using a distributed directory-based protocol.

The concept of directory-based cache coherence was first proposed by Tang [21]. However, we have had to extend it to work with distributed shared-memory [2, 16]. Each processing node in DASH has a directory memory corresponding to its portion of the shared physical memory. For each memory line, the directory stores the identities of all remote nodes caching that line. Using the directory memory, a node writing a location can send point-to-point invalidation or update messages

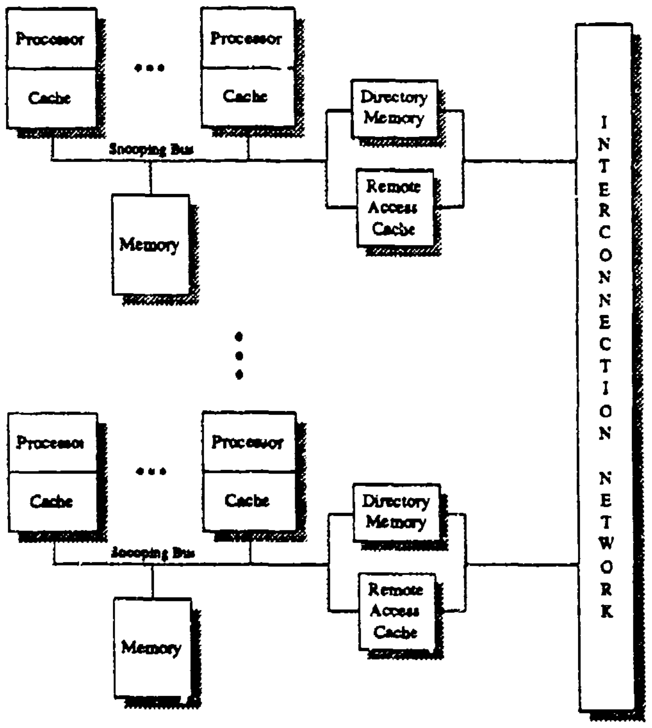

Figure 1: The DASH architecture consists of a set of processing nodes connected by a general interconnection network. Directory memory contains pointers for each memory line to the nodes currently caching that line. The remote access cache maintains state on the on-going remote requests.

to those nodes that are actually caching that line. This is in contrast to the invalidating broadcast required by the snoopy protocol. The scalability of DASH depends on this ability to avoid broadcasts. Another important attribute of the directory-based protocol is that it does not depend on any specific interconnection network topology. As a result, we can readily use any of the low-latency scalable networks, such as meshes or hypercubes, that were originally developed for distributed memory message-passing machines [6].

The rest of the paper is organized as follows. Section 2 describes the fundamental concepts of distributed directory based architectures. Section 3 describes the DASH architecture and then discusses how our design addresses some of the important problems faced by scalable shared-memory multiprocessors. Section 4 discusses implementation issues. Section 5 contrasts our approach with other recent proposals for scalable shared-memory multiprocessors. Finally, we present conclusions and summarize the current status of the prototype development.

2 Distributed Directory-Based Coherence

The key to scalability of any multiprocessor lies in the ability to scale the memory bandwidth available to each processor. DASH provides scalable bandwidth for private and shared data in local memory by distributing the physical memory among the processing nodes. To achieve the best performance, the operating system should allocate the local data of a process in the memory local to the processing node. For shared data accesses that must be serviced remotely, DASH uses a scalable interconnection network. The amount of global traffic on the network is reduced by providing coherent caches for shared data. Although the network bandwidth still needs to scale linearly, caching shared data reduces access latency, and allows a better match between local processing speed and the achievable bandwidth of the global interconnect.

While coherent caches are valuable, they generate coherence traffic and require extra hardware support. Focusing first on the coherence traffic generated by the directory-based protocol, we do not expect it to cause scalability problems. When a processor modifies a shared data location, the directory-based protocol sends point-to-point invalidation messages to all other processing nodes caching that location. Since invalidations need only be sent to those remote caches that had previously fetched the data, the invalidation traffic will be within some small constant factor of the regular data traffic. Since DASH provides scalable interconnection network bandwidth and scalable bandwidth to the directory memory (the directory memory and controllers are themselves distributed among the processing nodes), the invalidation traffic does not cause a scalability problem. In fact, due to the locality in shared data references [1], we expect each data item to be referenced several times before being invalidated, thus resulting in a significant reduction in global traffic.

The major scalability concern then becomes the amount of directory memory needed to keep track of the processing nodes that cache any given memory line. If the physical memory in the machine grows linearly with the number of processing nodes, then using a bit-vector to keep track of all nodes caching a memory line does not scale well. The total amount of directory memory needed is $P^2 \times K/L$ Mbits, where P is the number of processing nodes, K is the Mbytes of memory per processing node, and L is the cache-line size in bytes. Thus, the memory overhead of the simple directory scheme is P/L, which is proportional to the number of processing nodes and inversely proportional to the cache line size. Depending on the size of the machine, this growth with P may be easily tolerable or

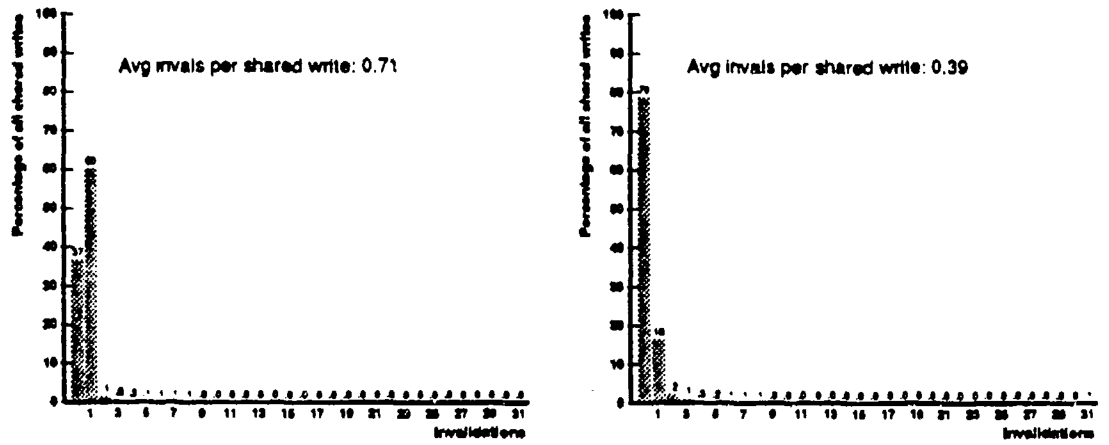

Figure 2: Cache invalidation patterns for MP3D and PTHOR. MP3D uses a particle-based simulation technique to determine the structure of shock waves caused by objects flying at high-speed in the upper atmosphere. PTHOR is a parallel logic simulator based on the Chandy-Misra algorithm.

can be remedied by keeping a lesser amount of information per memory line. For example, consider a machine with 32 processing nodes, each processing node with 8 processors, and assume that the cache line size is 32 bytes. For this 256 processor machine, the overhead of the directory memory is only 12.5%; comparable with the overhead usually tolerated to support an error correcting code on memory.

Our study of parallel applications has shown that in most situations, a memory location is cached by only a *small* number of processing nodes when written. Figure 2 shows the cache invalidation patterns as observed for the MP3D and PTHOR applications when running on a 32 processor machine. The graphs show that most writes cause invalidations to only a few caches [22]. Consequently, it is possible to replace the complete bit-vector in the directory by a small number of pointers, and to use broadcast invalidations in the unusual case when the number of pointers provided is too small [2]. We are currently investigating the performance of other possible schemes that limit the number of clusters that can cache a given line or use explicit overflow tables when the number of shared copies is greater than the number of pointers.

The above discussion assumes that the accesses are uniformly distributed across the machine. The uniform access assumption does not hold for highly contended synchronization objects[22]. DASH provides special extensions to the directory-based protocol to handle synchronization references, as discussed in Section 3.4.

The uniform access assumption also does not hold for references to heavily used shared data objects, for example, references to the variable holding the best solution in a branch-and-bound algorithm. This concentrated accessing of data from the memory of a single processing node over some short duration of time constitutes a *hot spot.* Under hot spot conditions, the memory and network bandwidth are reduced since the distribution of resources is not exploited as it is under uniform access patterns. DASH allows shared data to be cached and kept coherent, thus avoiding hot spots that may occur in other parallel machines that do not permit caching of writable shared data. In the example of the branch-and-bound algorithm, the bound is not updated very often in later stages of the search, but it is accessed for each move. In DASH, the bound will be cached in the processing nodes and there will be only limited hot-spotting when the bound is changed. In other architectures, the current bound will be a perpetual hot spot, unless the algorithm is restructured.

The issue of memory access latency becomes more prominent as an architecture is scaled to a larger number of nodes. There are two complementary approaches for managing latency: methods that reduce latency, and mechanisms that help tolerate large latencies. To provide a general-purpose architecture, DASH uses both of these approaches. Our primary focus has been to reduce the latency for accesses before trying to tolerate the latency. Techniques that depend on tolerating latency demand a large degree of parallelism to overlap communication and computation. For example, if the latency is l then $l \times P$ degrees of parallelism is needed to fully mask the latency. Since not all applications exhibit such large parallelism, we emphasize latency reduction techniques first.

Towards the goal of reducing latency for remote accesses, the hardware cache-coherence protocol permits processors to cache writable shared data. As will be discussed in section 3.1, the directory-based coherence protocol has been specially optimized for reducing latency. We expect the caching of shared data to significantly reduce the average latency for remote accesses because of the spatial and temporal locality in program accesses [1]. Furthermore, the distribution of shared memory among the processing nodes allows the programmer or compiler to exploit physical locality by placing data close to the node that will access it most.

3 The DASH Architecture

The DASH architecture has a two-level structure. At the top level, the architecture consists of a set of processing nodes (clusters) connected through a mesh interconnection network. In turn, each processing node is a bus-based multiprocessor. Intra-cluster cache coherence is implemented using a snoopy bus-based protocol, while inter-cluster coherence is maintained through a distributed directory-based protocol.

The cluster functions as a high-performance processing node. A bus-based cache protocol is chosen for implementing small-scale shared-memory multiprocessors because the bus bandwidth is sufficient to support a small number of processors. The bus-based design allows for fine-grain sharing within each cluster, and the directory-based scheme provides scalability at the inter-cluster level. The grouping of multiple processors on a bus within each cluster also enables the sharing of resources among these processors. Furthermore, this grouping reduces the directory memory requirements by keeping track of cached lines at a cluster as opposed to processor level.

3.1 The Coherence Protocol and Memory Hierarchy

The directory-based protocol implements an invalidation-based coherence protocol. A memory location may be in one of three states. It may be *uncached*, that is not cached by any processing node at all; *shared*, that is in an unmodified state in the caches of one or more nodes; or *dirty*, that is modified in a single cache of some node. The directory keeps the summary information for each memory line, specifying the clusters that are caching it.

An ideal system would have uniform, low latency access to all memory locations. However, this is impossible in any large-scale system. Memory hierarchy is a well-known technique to reduce memory

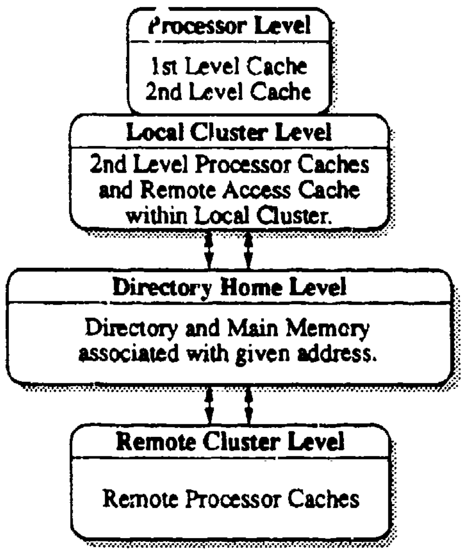

Figure 3: Memory hierarchy of DASH.

latency by taking advantage of physical locality.

The DASH memory system can be logically broken into four levels of hierarchy as illustrated by Figure 3. The level closest to the processor is the processor cache and is designed to match the speed of the processor. A request that cannot be serviced by the processor cache is sent to the second level in the hierarchy, the *local cluster* level. This level consists of other processors' caches and the remote access cache within the requesting processor's cluster. If the data is locally cached, the request can be serviced within the cluster, otherwise the request is sent to the *directory home* level. The home level consists of the cluster that contains the directory and physical memory for a given memory address. For some addresses, the local and home cluster are the same and the second and third level access occur simultaneously. In general, however, the request will travel through the interconnect to the home cluster. The home cluster can usually satisfy a request, but when the directory is in the dirty state, or in the shared state when the requesting processor requires exclusive access, the fourth, *remote cluster* level, must be accessed. The remote cluster level responds directly to the local cluster level while also updating the directory level.

To illustrate the directory protocol, we first consider how a processor read traverses the memory hierarchy:

Processor Level: If the location is present in the processor's cache, the cache simply supplies the data. Otherwise, the request goes to the local cluster level.

Local Cluster Level: In the event that the data resides in the caches within the cluster, the data is supplied to the requestor and no state change is required at the directory level. If the data is dirty in one of the processor caches, the remote access cache takes ownership of the location and supplies the requestor with the data. If the request must be sent beyond the local cluster level, then the remote access cache records the presence of the outstanding request. The remote access cache can then merge accesses to the same block made by other processors in the cluster. This merging reduces hot-spotting by filtering out all but the first request to a given block. If the local cluster fails to satisfy the request, the request is sent to the home cluster.

Home Directory Level: The home cluster examines the state of the memory location and can satisfy the request unless the location is dirty in a remote cluster. If not dirty, the data is sent to the requestor and the directory is updated to show sharing by the requestor. If the location is dirty remote, the request is forwarded to the remote cluster.

Remote Level: The remote cluster sends the data to the requestor, and writes back the data to the home cluster, which in turn updates main memory and the directory state appropriately. The remote cluster sends the data directly to the requestor, as opposed to routing it through the home, in order to reduce the latency involved in accessing remote dirty locations.

We now consider the sequence of operations that occurs when a location is written.

Processor Level: If the location is dirty in the writing processor's cache, the write can complete immediately. Otherwise, a read-exclusive request is issued on the local cluster to obtain exclusive ownership of the line and retrieve the remaining portion of the cache line.

Local Cluster Level: If the cache line is already dirty in one of the caches within the cluster, then the read-exclusive request can be serviced at the local cluster level by a cache to cache transfer. In this case, processors within a cluster can alternately modify the same memory location without any intercluster interaction. Otherwise, the ownership request needs to be sent to the home cluster.

Home Directory Level: The home cluster can immediately satisfy the ownership request if the memory location is not cached by any remote clusters. If the cache line is in shared state, all copies of the data must be invalidated. The contents of the directory memory indicate the processing nodes that need to be invalidated. Invalidation requests are sent to all clusters sharing the cache line except the requesting cluster. Meanwhile, the home cluster sends exclusive data to the local cluster which releases the requesting processor. When the processor re-issues the request it obtains the line and removes any shared copies from the other local processors' caches.

Remote Level: If the cache line is in dirty state in some remote cluster, the write request is forwarded to the remote cluster. The cluster sends the data to the requesting cluster and a directory update message to the home directory. The home sends an acknowledgment to the requesting cluster when the update is complete. If the home had indicated a shared state, then the remote clusters simply invalidate the cache line when they receive the invalidation request, and then send an acknowledgement to the requesting cluster.

When the writing processor receives all the invalidation acknowledgments, or the acknowledgment from the home directory that the state has been changed, it is guaranteed that the directory state is current and all copies of the old data have been purged from the system. If the processor delays completing the write until all acknowledgments are back, then the new value will become available to all processors at the same time. However, invalidations may involve round trip messages to multiple caches through the network, resulting in a long wait for the acknowledgment messages. We can achieve higher processor utilization if we allow the write to proceed immediately after the ownership request is satisfied. However, this can cause logical problems in the computational model for the multiprocessor. The next subsection describes how DASH provides minimum constraints on memory request ordering, while still providing a reasonable programming model to the user.

3.2 Memory Consistency

The level of memory consistency required by an architecture directly affects the amount of buffering and pipelining that can take place among memory requests. In addition, it has a direct effect on the complexity of the programming model presented to the user. Our goal in DASH is to provide maximum freedom in the ordering among memory requests, while still providing a reasonable programming model to the user.

We begin the discussion on consistency with some definitions. A write operation is considered globally performed when a read by any processor will return the value of that write or some write that logically follows that write [7]. A read operation is globally performed when the result of the read is bound to the value of some write that has been globally performed (including a write by another processor).

At one end of the consistency spectrum is the *sequential consistency* model [15], which requires the execution of the parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. Strong ordering [7] provides sufficient requirements for achieving sequential consistency. Sufficient conditions for strong ordering require a memory request to be globally performed before the next request by the same processor can be issued.

Strong ordering, while conceptually appealing, imposes a large performance penalty on writes. It prohibits the processor from completing a write until all related coherence traffic has been propagated throughout the system. For most applications, such a model is too strict and one can make do with a much weaker notion of consistency.

A weaker consistency model can be derived by relating memory request ordering to synchronization points in the program. As an example, consider the case of a processor updating a data structure within a critical section. If updating the data structure requires several writes, in the strongly ordered system, each write will stall until all other cached copies of that location have been invalidated. But these stalls are unnecessary as the programmer has already made sure that no other process can rely on that data structure being consistent until the critical section is exited. If we can identify the synchronization points, we need only ensure that the memory is consistent at those points. Therefore, the consistency can be relaxed by only requiring the operations to be globally performed before a release operation is observed. We call this model of consistency *release consistency* [10].

Such a scheme has two advantages. First, it provides the user with a reasonable programming model, since the programmer is ensured that when the critical section is exited, all other processors will have a consistent view of the modified data structure. Second, it permits the invalidations of different write operations to overlap, resulting in lower latencies for writes and higher overall performance. The disadvantage of the model is that all synchronization accesses must be identified by the programmer or the compiler.

The release consistency model is an extension to the *weak consistency* model [7]. The two models are similar in that memory needs to be consistent only at the time of synchronization. Release consistency, however, distinguishes between releasing (i.e. unlock) and non-releasing (i.e. lock) synchronization operations. Release consistency only delays the performing of a release operation until previous operations are performed. Weak consistency requires delaying of all synchronization operations.

DASH provides support for both strong and release consistency models. This is achieved by having a highly parallel read and write protocol that allows the memory to be inconsistent. However, the protocol has minimum constraints that, when coupled with *fence* instructions, can enforce different consistencies. A fence operation [4] stalls the execution of subsequent instructions until previous read and write operations are globally performed.

DASH supports the release consistency model in hardware. Read operations block until the load data is returned from the cache or the lower levels of the memory hierarchy. Write operations, however, are non-blocking. There is a write buffer between the first and second level caches. The write buffer queues up the write requests and issues them in order. Furthermore, the servicing of the write requests are overlapped. As soon as the cache receives the ownership and data for the requested cache line, the write data is removed from the write buffer and written into the cache line. The next write request can be serviced while the invalidation acknowledgments for the previous write operations filter in. Thus parallelism exists at two levels: the processor may execute other instructions while a preceding write operation takes place, and invalidations of multiple write operations may overlap.

DASH provides fence operations to allow emulation of other consistency models in software. There are two kinds of fence instructions. A *full fence* instruction that stalls a processor until all previous operations are globally performed. To implement strong consistency, it is sufficient for the processor to issue a full fence instruction after every write to a possibly shared location. DASH also has a *write fence* operation. The write fence instruction does not stall the processor; it queues up in the write buffer and a processor can continue executing instructions including issuing write operations into the write buffer. A write fence does not complete, however, until all previous writes from that processor have been performed. Release consistency can be implemented by queuing a write fence operation with the unlock operation in the write buffer. By the time the unlock operation is observed, all previous operations have been performed. The fence operations are implemented using counters that count the number of outstanding memory requests. The fence operation succeeds when there are no outstanding requests.

3.3 Memory Access Optimizations

The use of release consistency hides some of the memory latency for remote write operations. However, remote read operations still stall the processor for the entire duration of access. In addition, since read operations are blocking, no pipelining occurs on read requests initiated by the same processor. The weak constraints of release consistency allow us to remedy this by using a variety of prefetch and pipelining techniques. DASH provides several optimizations to hide the latency of memory operations.

3.3.1 Prefetch Operations

A prefetch operation is an explicit non-blocking request to fetch the data before the actual memory operation is issued. Hopefully, by the time the process needs that data value, the data has already been brought into its cache, thus hiding the latency of the regular blocking fetch. As a simple example of its use, a process wanting to access a row of a matrix stored in another processing node's memory can do so efficiently by first issuing prefetch reads for all cache blocks corresponding to that row.

The prefetch operation in DASH brings the data into the remote access cache. A subsequent memory access will then be serviced by this cache. The prefetched data is kept consistent like any other data. If another processor happens to write into the cache line before the data is accessed, the data will simply be invalidated. The prefetch will be rendered ineffective, but the program will execute correctly. This makes the programming and compilation task easier as prefetching a location that may be touched by another processor will not result in incorrect execution. Research on compiler optimizations to prefetch data is currently under way.

3.3.2 Update and Deliver Operations

In the prefetch scheme, by the time the consumer process realizes that it should prefetch some data it may be close to using it. Latency will not be effectively hidden in these cases. Likewise, if multiple consumers need the same item of data, the communication traffic can be reduced if data is multicast to all the consumers simultaneously. Therefore, DASH provides instructions that allow the producer to send the data directly to the consumers. Another advantage of these mechanisms is that the producing node knows exactly when the data is ready. The disadvantage of this scheme is that it may clutter the remote caches with data that the consumers are not yet ready to consume. The scheme also requires that the consumer addresses be supplied. There are two ways to specify the addresses. The *update-write* operation sends the new data directly to all processors that have cached the data; the *deliver* operation sends the data to explicitly specified destinations.

The *update-write* primitive updates the value of all existing copies of a data word. Using this primitive, a processor does not need to first acquire an exclusive copy of the cache line, which would mean invalidating all other copies. Rather, data is directly written into the home memory and all caches that are caching the line containing the update word. The caching state of the data is not modified. Update semantics are particularily useful for event synchronization, such as the release event from a barrier where an update-write can be used to release all waiting processors.

The *deliver* instruction explicitly specifies the destination clusters of the transfer. In this scheme, the producer first writes out the data to the cache line using regular write operations. At that time, any other cached copies are invalidated. The producer then issues a deliver instruction, explicitly giving the destinations in a bit vector. A shared copy of the cache block is sent to all the specified clusters, and the directory is updated with the fact that the data is now shared between the various clusters.

As an example of a use of the deliver instruction, consider the example of solving a system of linear equations using Gaussian elimination. Assume that the load is balanced by distributing the columns of the matrix among the processors. The pivot column needs to be accessed by all processors at the beginning of each iteration of the algorithm, and consequently a hot spot results. The *deliver* instruction can be used to multicast the pivot column to all the clusters working on the problem. This avoids hot-spotting, and the access to the column is significantly reduced since a processor needs only to fetch it from the remote access cache. No explicit synchronization is needed on completion of the deliver operation since it is simply a move of the cache block. Any early read of the block will be satisfied by the home or dirty cluster.

3.4 Support for Synchronization

The access patterns to locations used for synchronization are often different from those for other shared data [22]. For example, whenever a highly contended lock is released, a large number of waiting nodes rush to grab the lock. In case of barriers, a large number of processors need to be gathered and released efficiently. Such phenomena often cause hot spots in the memory system and can degrade the performance of large multiprocessors. Consequently, synchronization variables usually warrant special treatment.

DASH provides two extensions to the cache coherence protocol to support synchronization primitives: queue-based locks, and fetch-and-increment and fetch-and-decrement operations.

The *queue-based locks* provide for efficient implementation of locks and counting semaphores. Ideally, locks should meet the following criteria: (i) minimum amount of traffic generated by spin waiting, (ii) low latency release of a waiting processor, (iii) low latency acquisition of a free lock, and (iv) keeping the first three criteria true for high contention locks. Most architectures handle locks by providing an atomic test&set instruction and use a cached test-and-test&set scheme for spin waiting. This reduces the amount of traffic generated during spin waiting. While such systems are usually successful in satisfying the first three criteria for low contention locks, they fail for high contention locks. For example, assume there are N processors spinning on a lock value in their caches. When the lock is released, all N cache values are invalidated, and N reads are generated to the memory system. Depending on the timing, it is possible that all N processors come back to do the test&set on the location once they realize the lock is free, resulting in further invalidations and re-reads [3]. Such a scenario produces much unnecessary traffic and increases the latency in acquiring and releasing a lock.

The queue-based locks in DASH address this problem by keeping a list in directory memory of the nodes spinning in their cache for the lock. Using the directory memory for storing locks is natural since the directory is already set up to track nodes caching a data item. Furthermore, the directory is normally accessed in read-modify-write cycles that match the atomic updates necessary for locks. Caching a locked lock minimizes network traffic while a processor is spinning. If a lock is released which has waiting nodes, one node is choosen at random and is granted the lock. The grant request has update semantics in that a processor can acquire a granted lock by access on the local cluster, as opposed to invalidating the value in the cache and having the node re-fetch the location from the home. This scheme lowers both the traffic and the latency involved in releasing a node waiting on a lock. Informing only one processor of the release eliminates unnecessary traffic and reduces the latency that would be incurred if all waiting nodes were allowed to contend.

The *fetch-and-increment* and *fetch-and-decrement* primitives [8] provide atomic increment and decrement operations at the memory site. The value returned by the operations is the value before the increment or decrement. These operations have low latency and are useful for implementing several synchronization primitives such as barriers, distributed loops, and work queues. The serialization of these operations is small because they are done directly at the memory site. Serialization is based on how fast the memory can do the next atomic operation on the location once one is initiated. The low serialization provided by the fetch-and-increment operation is especially important when a large number of processors want to increment a location, as happens when getting the next index in a distributed loop. The benefits of the proposed operations become apparent when contrasted with the alternative of using a lock to achieve an atomic increment/decrement of a counter. The alternative requires acquiring the lock, reading the counter, incrementing it at the processor, writing it back, and releasing the lock. This results in more traffic, latency, and serialization.

The preceding has focused on hardware support for synchronization. We are also exploring software techniques that will help reduce latency, traffic, and hot spots. For example, the load on a single widely contended lock may be distributed to multiple locks through software techniques. For primitives such as barriers, it is possible to distribute the load by gathering and releasing through a tree, thus eliminating heavily contended locations. Using trees also reduces the latency for the gather and release stages. We are currently studying such trade-offs for other synchronization primitives.

4 The DASH Implementation

A prototype implementation of DASH is being developed in concert with the architecture. While we have developed a detailed software simulator of the system we feel that a full implementation is needed to uncover the issues in the design of a scalable cache coherence machine, to verify the feasibility of such designs, and to provide a platform for studying real applications running on a large ensemble of processors.

To manage the size of the prototype design effort, the processing node is based on a commercially available bus-based multiprocessor. The prototype system attempts to meet the primary goals of scalability and high performance within the constraints imposed by the base node. While these constraints have compromised performance in a few areas, the implementation should be indicative of the performance we would expect from an ideal DASH system. Since we believe that many of the architectural features under consideration can only be fully evaluated on actual hardware, we have included many of them in the prototype system. To aid in the evaluation of the system we are also providing dedicated hardware monitoring logic.

4.1 The DASH Processing Node

The prototype system uses the Silicon Graphics POWER Station 4D/240 as the base processing node. The 4D/240 system consists of four MIPS R3000 processors and R3010 floating-point coprocessors running at 25 MHz. Each R3000/R3010 combination is nominally rated at 20 VAX MIPS and 4 LINPACK MFLOPS. Each CPU contains a 64 Kbyte instruction cache and a 64 Kbyte write-through data cache. The 64 Kbyte data cache interfaces to a 256 Kbyte second-level write-back cache. The interface consists of a read buffer and a 4 word deep write-buffer. Both the first and second level caches are direct-mapped and support 16 byte lines. The first level caches run synchronous to the 25 MHz processors while the second level cache runs synchronous to the 16.67 MHz memory bus.

The second-level processor caches are responsible for bus snooping and maintaining consistency among all caches in the cluster. Consistency is maintained using the Illinois coherence protocol [17]. The main advantage of using the Illinois protocol in DASH is the cache-to-cache transfers specified in this protocol. While this does not improve performance for misses that would be serviced by local memory, it can greatly reduce the penalty for remote memory misses. The set of processor caches together with the remote access cache act as a cluster cache of remote memory. The memory bus (MPBUS) of the 4D/240 is a synchronous bus and consists of separate 32-bit address and 64-bit data

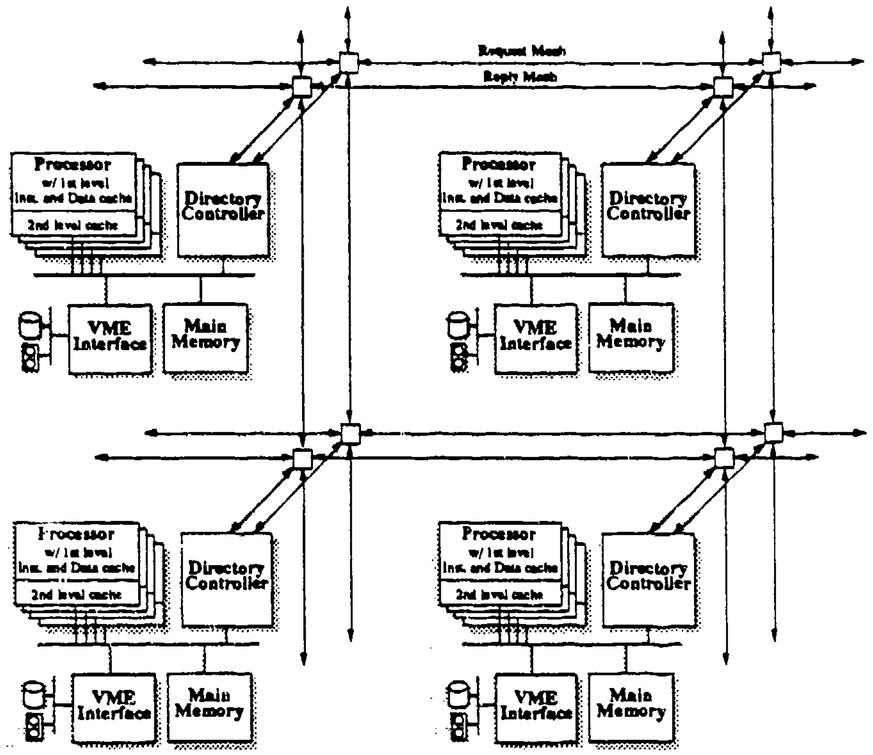

Figure 4: Block diagram of a 2 x 2 DASH system.

buses. The MPBUS is pipelined and supports memory-to-cache and cache-to-cache transfers of 16 bytes every 4 bus clocks with a latency of 6 bus clocks. Thus, the maximum bandwidth of the bus is 67 Mbytes/sec. While the MPBUS is pipelined, it is not a split transaction bus.

To use the 4D/240 in DASH, we have had to make minor modifications to the existing system boards and design a totally new directory controller board. The main modification to the existing boards is to add a bus retry signal that is used when a request requires service from a remote cluster. Retry, together with modifications to the bus arbitration logic to accept an arbitration mask from the directory board, allow the system to service remote requests without occupying the local bus for the entire duration of the request. Effectively, remote requests are handled using a split-transaction bus protocol. The new directory controller board consists of the directory memory, the intercluster coherence state machines, and the logic needed to support the interconnection network. The interconnection network consists of a pair of wormhole routed meshes, each with 16-bit wide channels. One mesh is dedicated to the request messages while the other handles replies.

A block diagram of four clusters interconnected to form a 2 x 2 DASH system is shown in Figure 4. Such a system could scale to support hundreds or thousands of processors, but the prototype will be limited to a maximum 4 x 4 configuration. This limit was dictated primarily by the physical memory addressability of the 4D/240 system, but still provides for systems of up to 64 processors that are capable of over 1200 MIPS and 250 MFLOPS.

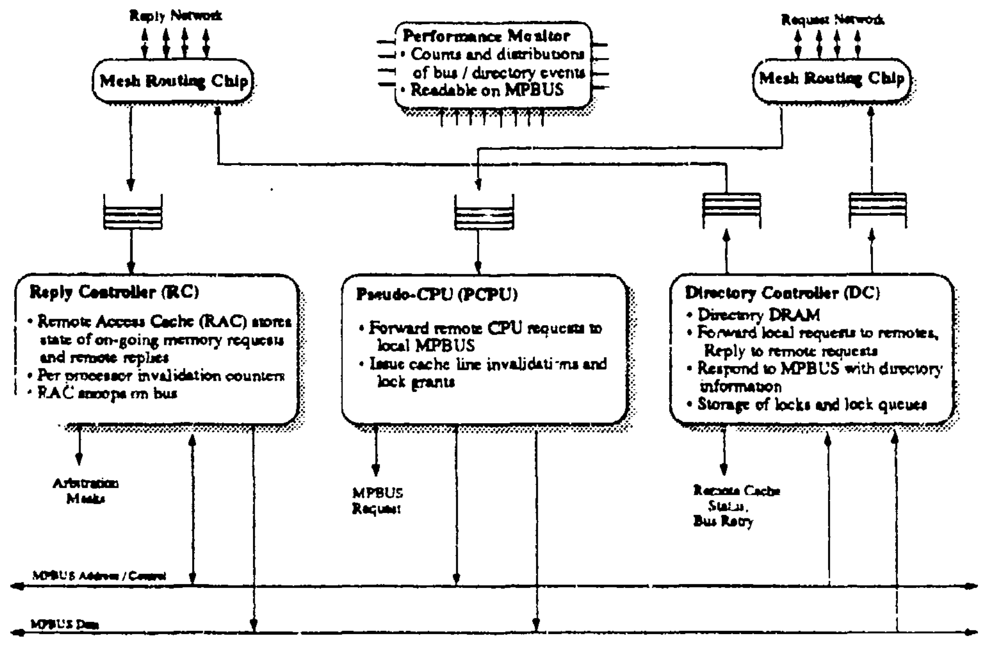
Reply Network
Mesh Routing Chip
Performance Monitor
• Counts and distributions of bus / directory events
• Readable on MPBUS
Request Network
Mesh Routing Chip
Reply Controller (RC)
• Remote Access Cache (RAC) stores state of on-going memory requests and remote replies
• Per processor invalidation counters
• RAC snoops on bus
Pseudo-CPU (PCPU)
• Forward remote CPU requests to local MPBUS
• Issue cache line invalidations and lock grants
Directory Controller (DC)
• Directory DRAM
• Forward local requests to remotes, Reply to remote requests
• Respond to MPBUS with directory information
• Storage of locks and lock queues
Arbitration Masks
MPBUS Request
Remote Cache Status, Bus Retry
MPBUS Address / Control
MPBUS Data

Figure 5: Directory board block diagram.

4.2 The DASH Directory Board

The directory board is responsible for implementing the directory-based coherence protocol and interconnecting the clusters within the system. A block diagram of the directory board is shown in Figure 5.

The directory board consists of five major subsystems. The *directory controller* (DC) includes the directory memory associated with the cachable main memory contained within a single cluster. It is also responsible for initiating out-bound network requests and replies. The *pseudo-CPU* (PCPU) is responsible for buffering incoming requests and issuing such requests onto the cluster bus. It mimics a CPU on this bus on behalf of remote processors except that responses from the bus are sent out by the directory controller. The *reply controller* (RC) tracks outstanding requests made by the local processors and receives and buffers replies from remote clusters using the *remote access cache* (RAC). The *network interface* and the local portion of the network itself reside on the directory card. The board contains *hardware monitoring logic* and miscellaneous control and status registers. The monitoring logic samples a variety of directory board and bus events from which usage and performance statistics can be derived.

Directory memory is accessed on each cluster bus transaction. The directory information is combined with the type of bus operation, the address, and the result of snooping other caches to determine what network messages and bus controls the DC will generate. The directory memory itself is implemented as a bit vector with 1 bit for each of the 16 clusters. While a full bit vector has limited scalability, it was chosen because it requires roughly the same amount of memory as a limited pointer directory given the size of the prototype, and it allows for more direct measurements of the caching behavior of the machine. Each directory entry also contains a single state bit that indicates whether the clusters have a read (shared) or read/write (dirty) copy of the data. The directory is implemented using DRAM technology, but performs all necessary actions within a single bus transaction.

4.3 Interconnection Network

As stated in the architecture section, the DASH coherence protocol does not rely on a particular interconnection network topology. However, for the architecture to be scalable, the network itself must provide scalable bandwidth. It should also provide low latency communication. The prototype system uses a pair of *wormhole* routed mesh networks to implement the interconnection network. The network chips are extended versions of the Caltech mesh routing chips [9] and have 16-bit wide data paths. Since wormhole routing allows nodes to forward a message after receiving only the first flit (flow unit) of the packet, the latencies in the network are small. We expect the latency through a node in the network to be approximately 40ns. The networks are asynchronous and self-timed. The speed of the network is limited by the round trip delay of the request-acknowledge signals. In DASH, we expect the flits to be transferred at 30 MHz, resulting in a total network bandwidth of 240 Mbytes/sec to each node.

An important constraint on the network is that it should be able to deliver request and reply messages generated by user programs and the coherence protocol without deadlocking. Most networks are guaranteed to be deadlock free if messages are consumed at the receiving node. In general, this assumption is not true for DASH. The processing nodes require the ability to generate an out-going message at the same time that they consume a in-coming message. For example, to consume a read request the home cluster must generate a reply message containing the data. Similarly, to consume a request for a location that is dirty in a remote cluster, the home cluster needs to generate a forwarding request to the remote cluster. This requirement adds the potential for deadlocks that consist of a sequence of messages that have circular dependencies through a node. These deadlocks are broken in two ways. The first relies on the fact that reply messages can always be consumed without dependency on out-going messages. Thus, having two independent mesh networks dedicated to request and replies guarantees that there will never be a request-reply dependency deadlock. Second, a timeout mechanism in the request network allows potential deadlocks in the request network (due to request-request dependencies) to be broken by rejecting requests through negative acknowledge reply messages.

4.4 Implementing Cache Coherence

A complete description of the implementation of the DASH coherence protocol is beyond the scope of this paper. The following examples, however, should give some insight into how the various sections of hardware interact to carry out the protocol. For a more detailed discussion see [16].

Figure 6 shows a simple read of a memory location whose home is a remote cluster and whose state in the directory is dirty in another cluster. The request is not satisfied within the local cluster and is sent to the home(1). The home cluster forwards the request to the remote cluster(2) where it is in dirty state. The dirty cluster then responds directly to the local cluster(3a) while simultaneously sending a sharing writeback to the home(3b) to update the directory and main memory.

Figure 7 shows the sequence of operations generated by a read-exclusive command to a remote address when the directory is in the shared state. The read-exclusive command originates with a

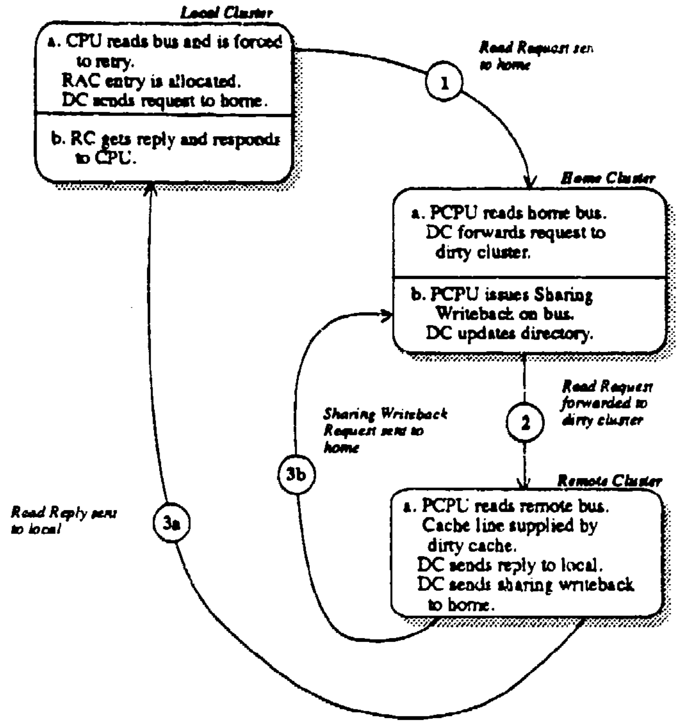
Local Cluster
a. CPU reads bus and is forced to retry.
RAC entry is allocated.
DC sends request to home.
b. RC gets reply and responds to CPU.
Read Request sen to home
1
Home Cluster
a. PCPU reads home bus.
DC forwards request to dirty cluster.
b. PCPU issues Sharing Writeback on bus.
DC updates directory.
Read Request forwarded to dirty cluster
2
Sharing Writeback Request sent to home
3b
Remote Cluster
a. PCPU reads remote bus.
Cache line supplied by dirty cache.
DC sends reply to local.
DC sends sharing writeback to home.
Read Reply sent to local
3a

Figure 6: Flow of a read request to remote memory.

processor write to a cache line that is either uncached by the processor or is held in the shared state. The write is buffered in the write-buffer, but the write-buffer must obtain exclusive access to the line before the write can complete. Assuming the line is in the shared state the request must first be sent to the home(1). The home directory replies to the request immediately(2a) while also sending invalidation requests to the clusters(2b) currently caching that line. Once the local cluster receives the data block from the directory, the release consistency model allows the processor to be granted exclusive ownership of the line, and for the write-buffer to retire the write. The RAC entry associated with the request persists until all the associated invalidation acknowledges (3) have been received.

An important feature of the coherence protocol is its forwarding strategy. If a cluster cannot directly reply to a given request, it forwards responsibility for the request to a cluster that should be able to respond. This minimizes the latency for a request, as the request is always forwarded to the location where the data is thought to be, which can then directly reply to the request. This technique also minimizes the serialization of requests since no cluster resources are blocked while inter-cluster messages are being sent. Forwarding allows the directory controller to work on multiple requests concurrently (i.e. make it multi-threaded) without adding the complexity of such multi-threading.

Performance of the various operations is closely related to the speed of the MPBUS and the latency of inter-cluster communication. For DASH we have attempted to minimize the number of bus and network transactions needed before the processor can proceed. Figure 8 shows the latencies for various memory operations assuming the system is unloaded.

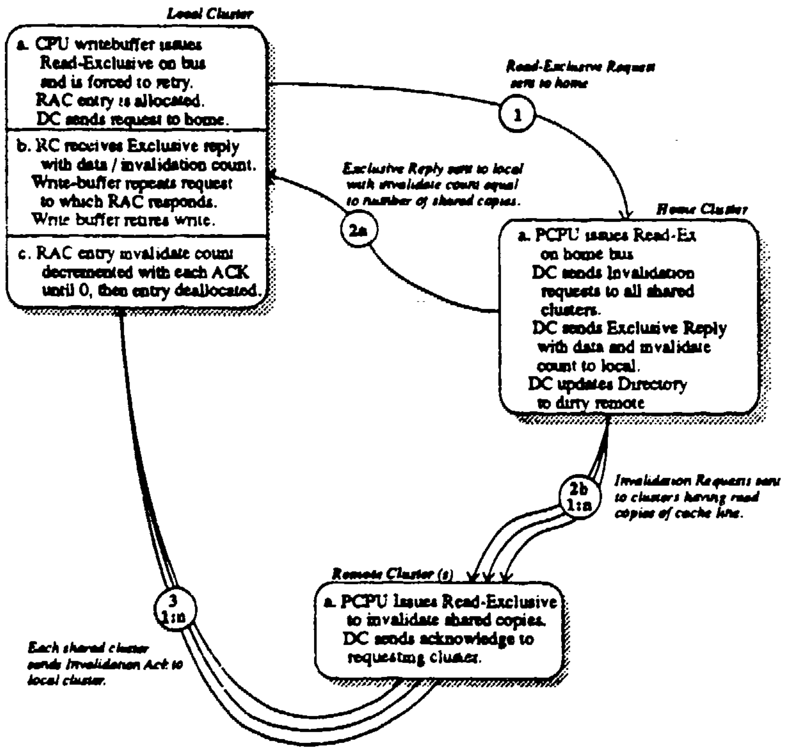
Local Cluster
a. CPU writebuffer issues Read-Exclusive on bus and is forced to retry. RAC entry is allocated. DC sends request to home.
b. RC receives Exclusive reply with data / invalidation count. Write-buffer repeats request to which RAC responds. Write buffer retires write.
c. RAC entry invalidate count decremented with each ACK until 0, then entry deallocated.
1
Read-Exclusive Request sent to home
2a
Exclusive Reply sent to local with invalidate count equal to number of shared copies.
Home Cluster
a. PCPU issues Read-Ex on home bus
DC sends Invalidation requests to all shared clusters.
DC sends Exclusive Reply with data and invalidate count to local.
DC updates Directory to dirty remote
2b 1:n
Invalidation Requests sent to clusters having read copies of cache line.
Remote Cluster (s)
a. PCPU Issues Read-Exclusive to invalidate shared copies. DC sends acknowledge to requesting cluster.
3 1:n
Each shared cluster sends Invalidation Ack to local cluster.

Figure 7: Flow of a read-exclusive request to remote memory.

Read Operations		Write Operations	
Hit in 1st Level Cache	1 pclock	Hit on 2nd Level Owned Block	3 pclock
Fill from 2nd Level Cache	12 pclock	Owned by Local Cluster	18 pclock
Fill from Local Cluster	22 pclock	Owned in Remote Cluster	57 pclock
Fill from Remote Cluster	61 pclock	Owned in Dirty Remote, Remote Home	76 pclock
Fill from Dirty Remote, Remote Home	80 pclock		
Fill operations fetch 16 byte cache blocks and empty the write-buffer before fetching the read-miss cache block. Write operations only stall the write-buffer, not the processor, while the fill is outstanding. Write delays assume Release Consistency (i.e. they do not wait for remote invalidates to be acknowledged).			

Figure 8: Latency for various memory system operations in number of processor clocks. Each processor clock for the DASH protoype is 40ns.

5 Alternative Approaches

In this section, we briefly compare DASH to other proposed scalable architectures that support cache-coherent shared-memory.

5.1 Encore GigaMax and Stanford VMP-MC

The Encore GigaMax architecture [23] and the Stanford VMP-MC project [5] both use a hierarchy-of-buses approach in building scalable shared-memory architectures. At the top level, the Encore GigaMax is composed of several clusters on a global bus. Each cluster consists of several processor modules, a cluster cache that caches remote accesses, and main memory connected by a cluster bus. Each processor module consists of several processors with private caches and a large shared second-level cache.

The VMP-MC machine is similar to the GigaMax in terms of the hierarchy of processors, caches, and buses. It is different, however, in that the physical memory is all located at the global level, and it uses a hierarchical directory-based coherence protocol. The clusters containing cached data are identified by a bit-vector scheme at every level instead of using snooping cluster caches. The VMP-MC also provides a lock bit per memory block that enhances performance for synchronization and explicit communication.

The global bus is the critical link in hierarchical bus-based machines. Unless an application's communication requirements match the bus hierarchy, or it has a very small sharing traffic requirement, the global bus will be a bottleneck. Both requirements are restrictive and limit the classes of applications that can be run on these machines. We feel the DASH architecture is more robust because: (i) it provides a scalable interconnect between the multiple clusters rather than a single global bus, and (ii) it makes effective use of this scalable point to point interconnection network through the directory-based cache coherence protocol.

5.2 Wisconsin Multicube

The Multicube architecture [13] employs a snooping cache system over a grid of buses. Each processor has a first-level cache that minimizes latency and a second-level cache, referred to as the snooping cache, that is designed to minimize bus traffic. Each snooping cache monitors two buses, a row bus and a column bus, in order to maintain data consistency among the snooping caches. In addition, there are modified line tables for snooping purposes. The memory of the machine is not distributed among the processors because the architects did not want to impose the notion of geographical locality onto the software. In support for synchronization, Multicube provides an efficient queuing mechanism for handling locks [12] that is similar to the approach taken in DASH.

We feel that a major drawback of Multicube's grid-of-buses design is that with powerful processors that need substantial silicon/board area, the buses will have to be long, and consequently will not run at very high speeds. In addition, since the second-level caches are large they will most likely be built using DRAM, and snooping on these DRAM caches will be slow, further limiting bus speed and increasing latency for accesses.

5.3 IEEE-SCI Protocol

The IEEE P1596, Scalable Coherent Interface [14], is similar to the DASH project in that it attempts to provide for scalable computer systems that rely on cache coherent processors. It differs from DASH in that it is a bus standard, not a complete system design. Thus, it only specifies the interfaces that each processor should implement, leaving open the actual processor design and interconnection between SCI nodes. This difference, together with the evolving nature of SCI make a detailed comparison impossible, but there are certain general areas where the two can be compared.

The SCI architecture, at a high level, is similar to DASH with each node having a processor, a portion of main memory and an interface to the interconnection network. The major difference is in how and where the directory information is maintained. In SCI, the directory is a distributed sharing list maintained by the caches. For example, if processors A, B, and C are caching some location, then the lines storing this location in these processor's cache form a doubly-linked list. Main memory holds only a pointer to the processor at the head of this linked list. In contrast, DASH places all directory memory with main memory.

The main advantage of the SCI scheme over DASH is that the amount of directory pointer storage grows naturally with the number of processors actually used in a system. In DASH, the maximum number of processors must be fixed beforehand, or the system must support some form of limited directory information. The primary disadvantages of the SCI scheme are due to the added complexity and latency required to manipulate the distributed directory entries. In addition to the multitude of interactions present in DASH, many more interactions are present in SCI as it lacks atomic updates to the directory entries. More importantly, the distributed nature of the SCI directory entries inherently increases the latency of operations. Again, while a complete evaluation is not possible at this time, we feel that minimizing latency is critical, and that the problems of maintaining the pointers at memory can be solved through schemes using a limited number of directory pointers.

6 Summary and Status

In this paper we have described the design and implementation decisions for DASH, a scalable shared-memory multiprocessor with coherent caches. The key concepts in achieving scalability are the use of a distributed directory-based cache coherence protocol, distributed memories and directories, and a scalable interconnection network. The design optimizes along the dimensions of scaling the memory bandwidth and reducing the latency of remote memory accesses. We have also discussed several other performance enhancements for DASH. These include mechanisms for reducing and tolerating latency such as prefetch, the release consistency model and synchronization primitives such as queue-based locks.

We are actively proceeding with building a prototype of DASH. At the time of writing, we have completed the design of the coherence protocol and the associated state machines, and we have identified the modifications necessary to the existing Silicon Graphics hardware. We are currently at the schematic entry phase for the directory board. The interconnection network has already been designed and we have recently received the network chips back from fabrication. Our goal is to have a prototype with 16 processors up and running by the end of summer of 1990.

In addition to the task of building the prototype, we are putting substantial effort in developing trace generation and architecture simulation tools. Tango [11], a powerful multiprocessor emulator that generates address traces, is coupled with simulators for DASH and other architectures to provide detailed performance data. We are also using the DASH simulator for validating and verifying the directory-based coherence protocol. On the software side, we are working on a parallelizing compiler, developing software macro libraries, and providing an operating system for DASH. The DASH parallelizing compiler expands on the work previously done at Stanford on partitioning and scheduling [19]. The software macro libraries will incorporate the special features of DASH in providing synchronization and scheduling functions. The DASH operating system will be either an SGI's UNIX System V.3 or a version of MACH tailored to our machine.

To help evaluate and focus our parallel software and hardware efforts, we are also working on several parallel applications. The applications include distributed-time digital logic simulation, solution of large sparse systems of equations, standard cell routing, ocean simulation, and particle simulation for high-speed flight. To support DASH application development and optimization, we are developing software performance monitoring and analysis packages that couple with our hardware monitoring subsystem to provide the programmer with effective performance information about an application.

7 The DASH Team

There are a number of graduate students and faculty working on the DASH project. The Ph.D. students are: Daniel Lenoski, Kourosh Gharachorloo, and James Laudon (DASH architecture and hardware design); Wolf-Dietrich Weber (invalidation patterns, DASH simulator, scalable directories); Helen Davis and Stephen Goldschmidt (trace generation tools, synchronization patterns, locality studies); Margaret Martonosi and Tom Chanak (mesh routing chip simulation and design); Richard Simoni (synthetic load generator and directory studies); Truman Joe (SGI board modifications and simulator for Encore Gigamax); Josep Torrellas (sharing patterns in applications); Eric Williams and Raul Izahi Lopez (SGI board modifications); Todd Mowry (evaluation of prefetch operations); Edward Rothberg, Jaswinder Pal Singh, Larry Soule (applications and algorithm development);. Bruce Kleinman, a master's student, is working on verification of the DASH coherence protocol. Research engineer David Nakahira has also contributed to the hardware design. The faculty associated with the project are Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam.

8 Acknowledgments

We would like to thank Wolf-Dietrich Weber, Helen Davis, Rich Simoni, Steve Goldschmidt, Margaret Martonosi, Jaswinder Pal Singh, Paul Calder, and Dror Maydan for their comments on the paper. This research was supported by DARPA contract N00014-87-K-0828.

References

[1] Anant Agarwal and Anoop Gupta. Memory-reference characteristics of multiprocessor applications under MACH. In *Proceedings of SIGMETRICS 1988*, May 1988.

[2] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual Internanonal Symposium on Computer Architecture*, pages 280–289, June 1988.

[3] Thomas E. Anderson. The performance implications of spin-waiting alternatives for shared-memory multiprocessors. In *Proceea.:¬gs of the 1989 International Conference on Parallel Processing*, pages II,170–174, August 1989.

[4] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 processor-memory element. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 782–789, 1985.

[5] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Multi-level shared caching techniques for scalability in VMP-MC. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 16–24, June 1989.

[6] William J. Dally. Wire efficient VLSI multiprocessor communication networks. In *Stanford Conference on Advanced Research in VLSI*, 1987.

[7] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[8] J. Elder, A. Gottlieb, C. K. Kruskal, K. P. McAu`iffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson. Issues related to MIMD, shared-memory computers: The NYU Ultracomputer approach. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 126–135, June 1985.

[9] Charles M. Flaig. VLSI mesh routing systems. Technical Report 5241:TR:87, California Institute of Technology, May 1987.

[10] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. Technical Report CSL-TR-89-405, Stanford University, 1989.

[11] Stephen R. Goldschmidt. Tango tutorial. Technical Report to appear, Stanford University, 1989.

[12] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–73, April 1989.

[13] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, June 1988.

[14] P1596 Working Group. P1596/Part IIIA - SCI cache coherence overview. Technical Report Revision 0.33, IEEE Computer Society, November 1989.

[15] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[16] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. Technical Report CSL-TR-89-404, Stanford University, 1989.

[17] Mark S. Papamarcos and Janak H. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, June 1984.

[18] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

[19] Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceeding of the SIGPLAN '86 Symposium on Compiler Construction*, pages 17–26, July 1986.

[20] G. E. Schmidt. The Butterfly parallel processor. In *Proceedings of the Second International Conference on Supercomputing*, pages 362–365, 1987.

[21] C. K. Tang. Cache design in the tightly coupled multiprocessor system. In *AFIPS Conference Proceedings, National Computer Conference, NY, NY*, pages 749–753, June 1976.

[22] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[23] Andrew W. Wilson, Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 244–252, June 1987.

Ninth International Conference on Computing Methods in Applied Sciences and Engineering
Paris, France, January 29 - February 2, 1990

Compiler Optimizations for Superscalar Computers

Monica S. Lam

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Capable of executing multiple scalar operations per cycle, a superscalar architecture can parallelize not just vectorizable programs, but also code containing recurrences and data dependent control flow. This paper presents an overview of the compiler optimizations that are crucial in harnessing the computation power of superscalar machines. These optimizations include high-level loop transformations to find parallelism and improve the efficiency of caches, software pipelining and hierarchical reduction techniques for scheduling instructions, and modulo variable expansion for assigning registers.

Recent advances in hardware technology have made superscalar architectures amenable to single-chip implementations. The combination of cheap hardware to provide a high raw computing power and sophisticated compiler technology to effectively use the parallelism can produce extremely low-cost, high-performance workstations that are easily accessible to the general scientific and engineering community.

1. Introduction

A superscalar computer is a uniprocessor that can execute two or more scalar operations in parallel. The operations are individually specified in the object code; this is distinct from vector machines which expand vector instructions into series of parallel operations. The parallelism of a vector instruction is defined for each vector machine at machine design time; on a superscalar machine, a parallel execution schedule is created uniquely for each program, by either hardware or software. As a result, superscalar machine organizations are more versatile and effective in using the hardware resources in the system.

Superscalar machines existed long before the term was coined. The IBM Stretch [5], the CDC 6600 [24] and the IBM 360/91 [2] are all superscalar architectures that can execute multiple operations in parallel. These machines all implement a sequential instruction set with hardware that schedules the instructions dynamically. Besides hardware, software has also been used for instruction scheduling. Epitomizing the class of superscalar machines that rely on software for scheduling instructions is the

VLIW (Very Long Instruction Word) architecture [13]. Each wide instruction word explicitly specifies the operations to be executed in parallel. Examples of such machines include the Multiflow's Trace machines [8], the Carnegie Mellon's processors for the Warp systolic array [3] and the Cydrome's Cydra 5 [9]. The recent hardware technology advances have made software scheduled superscalar architectures amenable to single-chip implementations. A follow-on of the Warp processor, the Carnegie Mellon and Intel's iWarp processor integrates high-performance computation and systolic communication in a single component [6]. The Intel's i860 is a single-chip microprocessor that can perform up to 100 million floating-point per seconds (MFLOPS) using a dual-instruction word format [17].

The development of the recent superscalar architectures presents an exciting prospect to the engineering and scientific community. As technology improves, the superscalar processor performance is expected to grow. The superscalar processor provides a more flexible form of instruction parallelism in a low-cost package. The impact is that high computing power can be easily provided in a low-cost desktop workstation that is widely accessible to engineers and scientists. The high-level of integration also makes these scalar processors a useful building block for large-scale multiprocessing, thus delivering an aggregate computation bandwidth higher than ever before.

The parallelism of a superscalar machine may be managed in hardware or software. The hardware approach schedules the instructions dynamically, thus hiding parallelism from the architecture. The instruction set architecture can therefore be made compatible with that of an existing sequential machine. Run-time scheduling, however, requires more hardware logic, which may result in a slower clock cycle or longer latency in instruction execution. In the software approach, the parallelism is exposed at the architecture level, and the compiler is responsible for specifying the parallel operations to execute. By analyzing the entire program statically, the compiler can exploit higher level program semantics and rearrange the code globally to derive a better schedule.

To harness the raw computing speeds of software-scheduled superscalar processor in applications, compiler technology is crucial. The compiler hides the parallelism from the programmer, so the programmer can develop applications easily using a high-level sequential language. This approach has the additional advantage that the same sequential programs can now easily be ported to other current and future machine architectures.

In this paper, we first describe the characteristics of the superscalar architecture and the issues in compiling code for such machines. We then present a set of compiler optimizations, showing how the functionality of the processor can be used in programs. We then close with a discussion on the performance of these superscalar machines.

2. Superscalar Architectures

Common to all superscalar processors is the presence of parallel and/or pipelined functional units. Like any machine that employs parallelism and pipelining, a program running on a superscalar seldom achieves the peak computation rate of the machine. If a superscalar processor has *n* functional units, or a functional unit with *n* pipeline stages, *n* independent operations must be present at all times to utilize the machine fully. If no parallelism is found, the machine may operate at $1/n$th of the peak rate. Therefore, for a superscalar to be effective, it is important that the scheduler can find enough independent operations to execute in parallel.

Before we discuss the scheduling techniques, let us first take a look at the fundamental limit the hardware imposes on the execution speed of a program. Even if there are enough independent operations, the full computation power of a superscalar may not be brought to bear on an application because of specialization. The processor typically consists of a set of specialized functional units, some memory access units, possibly different arithmetic units, and an instruction branch control unit. For example, a program that requires no multiplications will not be able to take advantage of the multiplication unit on the processor.

The hardware of a system is typically designed such that the distribution of the computational units matches the distribution of operations in a typical program. From the statistics of a large set of numerical applications [18], we have observed that there are about as many floating-point arithmetic operations as memory operations. About 60% of the memory operations are read operations, and about 70% of the floating-point operations are additions. On a machine that can execute one memory read, one memory write, one floating-point addition, and one floating-point multiplication in a single cycle, the adder is often the critical resource and is followed by the memory read unit.

Besides the utilization of the functional units in a processor, it is also important to consider the memory subsystem. To support a high computation bandwidth, a processor must also have a similarly powerful memory subsystem. For a vector machine, the more restricted mode of operation permits the use of vector registers and efficient block transfers between the memory subsystem and the registers. Being able to support a less regular form of parallelism, a superscalar architecture requires a more flexible memory system. The concept of memory hierarchy has been shown to be useful in reducing the average access latencies for general-purpose machines. A cache can also reduce the number of memory accesses which can be important in a multiprocessor environment.

Unfortunately, a cache sometimes behaves rather poorly for numerical code. Because of the large data set used, data brought into the cache may be flushed out before they are reused. The cache hit rate can fluctuate widely depending on, for example, whether a matrix operand is in the cache. This may greatly affect the overall speed obtained due to the large difference between cache and memory speeds. While a cache is normally transparent to compilers for general-purpose programs, it is beneficial to optimize the cache behavior in superscalar compilers.

In many ways, a superscalar compiler faces similar issues as those of a vectorizing compiler. The compiler must extract parallelism from sequential programs and try to use the parallel, specialized functional units effectively. The compiler must also manage the cache; this is analogous to the management of vector registers in vectorizing compilers. Though the issues are similar, a superscalar machine presents new challenges to compiler optimization. The parallelism must be managed at the scalar operation level and the parallelism exploitable is not regular like vector instructions.

3. Overview of Compiler Techniques

There are two levels of compiler optimization: the loop level and the instruction level. The loop level involves higher level transformations on the loop structure. These transformations are useful both for bringing parallelism to the innermost loop as well as improving data locality. This high-level restructuring prepares the loop for low-level instruction scheduling.

The instruction level optimization consists of instruction scheduling and register assignment techniques. The scheduling problem is to find the shortest instruction schedule that satisfies the constraints imposed by the machine resources and the program semantics. In particular, since most of the computation time is spent on innermost loops, it is important to schedule such loops efficiently. *Software pipelining* is a scheduling technique that exploits the repetitive nature of innermost loops to generate highly efficient code for processors with parallel, pipelined functional units [19, 22, 25]. Another code scheduling technique used with software pipelining is *hierarchical reduction*, a technique that abstracts control constructs as operations in a basic block, so the same scheduling algorithms can be applied to within and across basic blocks. For example, using hierarchical reduction, software pipelining can be applied to all innermost loops, including those containing conditional statements. Hierarchical reduction makes it possible to obtain a consistent performance improvement for many more programs. Interacting with code scheduling is register assignment. When the same register is assigned to different variables, their uses must be serialized, thus constraining the parallelism in the computation. Therefore, the register assignment must also be considered hand-in-hand with instruction scheduling.

In the following, we first present an overview of the analysis techniques necessary to support both loop level and instruction level parallelism. We then discuss each of the optimizations: loop level transformations, software pipelining, hierarchical reduction and register assignment.

Program semantics produces two kinds of constraints: control dependences and data dependences. A conditional branch instruction must first be executed to determine the instruction to execute next. This sequencing constraint is known as *control dependence*. An operation cannot execute until all its operands are produced. This sequencing constraint is known as *true data dependence*. To ensure that a read operation always reads the latest value produced, the order of the write operations on the same location must also be observed. This sequencing constraint is known as *output dependence*. Furthermore, since a data location may hold different values at different times, a value must not be overwritten before its use. This form of data dependence is known as *anti-dependence*.

The compiler must first extract dependence constraints from the program. The analysis algorithms are similar to those previously used for vectorizing and concurrentizing compilers. The control dependence can either be obtained through analysis of the flow graph [11], or simply retained from the syntactic control structure of the program [16]. For data dependence, since array references are very common in numerical code, it is important to determine if two array references can refer to the same location, and thus may share a dependence relationship between them. Various dependence tests have been proposed for disambiguating between array references whose indices are an affine function of loop indices [1, 4, 27].

The dependence information was used previously only for source-to-source loop transformations. For a superscalar machine, this information is used at both the loop and instruction level. In the compiler currently developed at Stanford, data dependence is captured in an intermediate representation that supports loop level transformations, and this same information can be used in the code generation phase.

4. Loop Level Transformations

High level code transformations are useful in bringing parallelism into the innermost loop, as well as improving the efficiency of the caches. Consider the simple example of a matrix multiplication:

```
FOR i := 0 TO n-1 DO
   FOR j := 0 TO n-1 DO
       FOR k := 0 TO n-1 DO
           C[i,j] := A[i,k]*B[k,j]+C[i,j];
```

The result of one addition is used by the addition in the next iteration of the loop. The additions must therefore execute sequentially; with an *n*-stage pipelined adder, an iteration takes at least *n* clocks. The multiplications, being independent, can execute in parallel with the additions. (Unlike a vector machine, a superscalar machine can execute some instructions in parallel even for recurrences.) To further increase the utilization of the machine, the compiler must perform higher level transformations so as to expose more parallelism in the innermost loop to the instruction scheduler. In this example, if the inner two loops are interchanged as follows:

```
FOR i := 0 TO n-1 DO
   FOR k := 0 TO n-1 DO
       FOR j := 0 TO n-1 DO
           C[i,j] := A[i,k]*B[k,j]+C[i,j];
```

The iterations in the innermost loop are now independent; as many iterations as necessary can execute in parallel to fully utilize the hardware resources of the machine. Therefore, when the innermost loop does not contain enough parallel operations to keep the hardware resources busy, high level transformations, similar to those used in vectorizing and parallelizing compilers, should be applied.

For superscalar machines with caches, high level transformations can also be used to improve overall performance by reducing the cache miss rate. Consider a machine whose cache is relative small in comparison with the matrix size. The objective of the optimization is to minimize memory accesses by reusing data in the cache as much as possible. In the optimized program above, the innermost loop accesses rows k and i of matrices B and C, respectively. The same row of C is used in the next outer loop, but the B data will not be reused until the next iteration in the outermost loop. If the data size is large compared to the cache, even the C data may not be in the cache, let alone the B data. Maximum reuse is obtained if we can block, or tile, the computation as follows:

```
FOR ii := 0 TO n-1 BY b DO
   FOR kk := 0 TO n-1 BY b DO
       FOR jj := 0 TO n-1 BY b DO
           FOR i := ii TO min(ii+b-1, n) DO
               FOR k := kk TO min(kk+b-1, n) DO
                   FOR j := jj TO min(jj+b-1, n) DO
                       C[i,j] := A[i,k]*B[k,j]+C[i,j];
```

Each of the matrix elements brought into the cache is reused b times before it is removed from the cache. The value of b is chosen to maximize the cache utilization.

Previous research on data locality has provided ways to predict the cache behavior of a loop nest. Gannon et al. [14] use *uniformly generated references* to find where locality exists in a nesting of loops. They also discuss how to choose which array elements should go into the cache for a given loop. Porterfield [21] estimates cache behavior for a loop nest assuming that the cache uses the least recently used replacement policy, and may block a loop if the cache cannot hold all the data in an iteration. Gannon et al.'s and Porterfield's estimates can be used to evaluate the data locality of entire loop nests obtained by different sets of transformations.

Loop transformations beneficial to data locality and parallelism for superscalar machines include loop interchange, reversal, skewing and tiling. Wolf and I have developed an efficient algorithm to search through the space of these transformations and generates code that displays data locality and parallelism in the innermost loops [26]. We reduce the optimization problem to placing the maximum number of loops identified to carry locality in the innermost tile. Using this goal and the legality considerations of tiling, we can significantly prune the search space to find the best set of transformations. How tiling improves data locality has been illustrated by the example above. The conditions that made tiling legal in the first place guarantee both coarse and fine grain parallelism within a tiled loop. Therefore, by tiling the loops, we generate code that exhibits both data locality and parallelism.

5. Software Pipelining

After performing the high-level transformations, the compiler can then apply the instruction level optimizations. The basic technique for obtaining parallelism is software pipelining. Let us introduce the concept of software pipelining by way of an example. Suppose we have a machine that can perform one load, one store, and initiate a 7-stage pipelined floating operation in one instruction, and suppose the code we want to execute is:

```
FOR i := 1 TO n DO
   A[i] := A[i]+1.0;
```

Assume for now that we can generate the addresses for the loads and stores in parallel with the rest of the computation; the specifics of this topic will be discussed in Section 7. The most compact instruction sequence to execute a single iteration of this loop is given in Figure 5-1. The operation `BLoop 1` branches back to label 1 if there are more iterations to execute. The schedule is sparse due to the heavy pipelining in the data path. (For machines with hardware interlocks, the nop instructions are used only at code scheduling time; they are omitted when the code is emitted.) If we simply iterate this schedule, the throughput of the loop is only 1 iteration every 9 clock ticks, and no resources are used more than 1/9th of the time.

```
1: LD
   FADD
   nop
   nop
   nop
   nop
   nop
   nop
   ST    BLoop 1
```

Figure 5-1: Object code for one iteration in example program

Techniques such as trace scheduling [12] depend on loop unrolling to generate enough parallel instructions to sch[illegible]uppose the loop body of the example is unrolled 9 times, the optimal schedule of the body of [illegible]d loop is shown in Figure 5-2. (This instruction sequence assumes that the number of iterations is divisible by 9.) Each row in the figure corresponds to operations in an instruction, and each column corresponds to the computation of one iteration of the loop in the source program. Unrolling the loop 9 times improves the throughput to 9 iterations every 17 clocks. From the figure, it is clear that unrolling an additional iteration will only lengthen the schedule by one clock. This can be kept up until the iterations run out. A loop unrolled *u* times will have a throughput rate of *u*/(*u*+8) iterations per clock, while the ideal throughput is 1 iteration per clock.

```
1:  LD
    FADD  LD
          FADD  LD
                FADD  LD
                      FADD  LD
                            FADD  LD
                                  FADD  LD
                                        FADD  LD
    ST                                        FADD  LD
          ST                                        FADD
                ST
                      ST
                            ST
                                  ST
                                        ST
                                              ST
                                                    ST   Bloop 1
```

Figure 5-2: Optimal schedule for nine iterations

Although the schedule improves as we unroll more iterations, code expansion limits the degree of unrolling. Unrolling can therefore overlap only a small finite number of iterations; all the unrolled iterations must complete before the program branches back to another set of contiguous iterations. On a vector machine, such a loop maps directly into a vector instruction; a vector instruction can continually overlap operations from successive iterations to deliver a throughput of one iteration per clock cycle.

Software pipelining can achieve the same kind of performance obtained with vector instructions by continually overlapping operations from different iterations of a loop. The software pipelined program for the example above is shown in Figure 5-3. Code generated by software pipelining is compact. The code in the figure assumes that there are at least nine iterations in the loop. The first eight instructions constitute the *prolog* where more and more iterations of the loop start to execute. The *steady state* is reached after eight instructions, and is repeated until all iterations have been initiated. In the steady state, nine iterations are in progress at the same time, with one iteration starting up and one finishing off every clock. On leaving the steady state, the iterations currently in progress are completed in the *epilog*, the 10th through 17th instructions. This program achieves the *optimal* computation time by executing *n* iterations in *n*+8 clock ticks, where *n* is the number of iterations in the loop.

Software pipelining is different from loop unrolling in that a source iteration may span one or more iterations in the object code. If the machine contains pipelined functional units, the pipeline stages need not be emptied at iteration boundaries. In the example above, seven additions initiated in seven different

```
     LD
     FADD  LD
           FADD  LD
                 FADD  LD
                       FADD  LD
                             FADD  LD
                                   FADD  LD
                                         FADD  LD
1:   ST                                        FADD  LD    BLoop 1
           ST                                        FADD
                 ST
                       ST
                             ST
                                   ST
                                         ST
                                               ST
                                                     ST
```

Figure 5-3: Program of a software pipelined loop

iterations execute in parallel. The hardware pipelines are filled and drained only once on entering and exiting the loop, respectively. Software pipelining is especially beneficial for machines with high degrees of parallelism and specialization. The results are that optimal throughput can be achieved, and achieved with an extremely compact program.

5.1. The Problem

In this section, we first concentrate on the scheduling of loops containing a single basic blocks. Extending software pipelining to other loops is discussed with hierarchical reduction in the next section. The primary goal of software pipelining is to maximize the throughput in executing the iterations; it does not matter if the execution time of individual iterations is lengthened. Its secondary goal is to keep the code size down. In other words, the schedule must have a short steady state so that it can be captured in a relatively succinct code sequence. The problem is thus formulated as finding a common schedule for all iterations of the source loop, such that successive iterations are initiated with a constant interval, and the objective is to minimize this interval. In the example above, the schedule of an iteration is given in Figure 5-1, and the *iteration initiation interval* is one.

Software pipelining was originally derived from a technique for scheduling hardware pipelines, where the problem was formulated as inserting delays between hardware units to increase the overall throughput of the system [20]. New input is accepted by the hardware pipeline at regular periodic intervals. The software analog is to schedule operations within an iteration such that the iterations can be pipelined to yield optimal throughput.

Software pipelining has been used in compilers for several different architectures. The algorithm was first used in the ESL polycyclic architecture [22]. The polycyclic machine uses a specialized crossbar to simplify the scheduling problem for a subset of loops [23]. The same concept is also implemented in Cydrome's Cydra 5 [9]. Software pipelining is also used in the compiler for the FPS-164 machine [25]. The FPS-164 does not have any specialized support for software pipelining, and software heuristics are used to schedule the loops. We improved upon the FPS heuristics, especially in the algorithm for scheduling recurrences, and implemented them in our compilers [7, 19] for the Carnegie Mellon's Warp and iWarp machines. Eisenbeis et al. applied software pipelining to the problem of scheduling vector instructions, and implemented a compiler that generates software pipelined vector code for the Cray-2 architecture [10].

Let us first describe some of the fundamental limits in scheduling a loop. There are two kinds of constraints: resource and precedence constraints.

Resource Constraints. Suppose a machine has $m(r)$ units of resource r, and an iteration of a loop requires $n(r)$ units of resource r, then a pipelined loop cannot execute faster than the rate of at most one iteration every

$$\max_r \left\lceil \frac{n(r)}{m(r)} \right\rceil$$

cycles. This equation reconfirms the notion that it is harder to fully utilize highly specialized functional units and the computation rate is limited by the resource with the highest demand.

In software pipelining, we must ensure that the resource commitment in each clock cycle of the steady state does not exceed the available resources. The resource usage of the steady state can be represented by a *modulo resource reservation table* whose ith entry contains the sum of the resources used in cycles $i, i+s, i+2s, \ldots$ of the schedule of an iteration, where s is the initiation interval of the loop.

Precedence constraints. While recurrences limit the throughput of the computation, a superscalar, unlike a vector machine, can often still find some parallelism in such loops. Consider the following example:

```
FOR i := 1 to 100 DO
    a := a + 1.0;
```

We must first read a before we write back into a in the same iteration, which in turn must precede the read operation in the next iteration. The flow graph representing the above example, assuming a seven-staged addition, is shown in Figure 5-4. Each edge is labeled by the number of iterations the dependence crosses and the delay between them. As shown in the figure, inter-iteration data dependences may introduce cycles into the precedence constraint graph. The precedence constraints in Figure 5-4 impose a delay of 9 clock ticks between load operations from consecutive iterations. That is, loops cannot execute at a rate greater than one iteration every 9 clocks.

We define the minimum delay, d, and minimum iteration difference, p, of a path to be the sum of the minimum delays and minimum iteration differences of the edges in the path, respectively. If we let c denote a cycle in the graph, the rate at which the iterations can be executed is one iteration every

$$\max_c \left\lceil \frac{d(c)}{p(c)} \right\rceil$$

cycles.

The maximum of the two bounds determined by resource and precedence considerations establishes a lower bound on the initiation interval. Therefore, a schedule that pipelines with an initiation interval

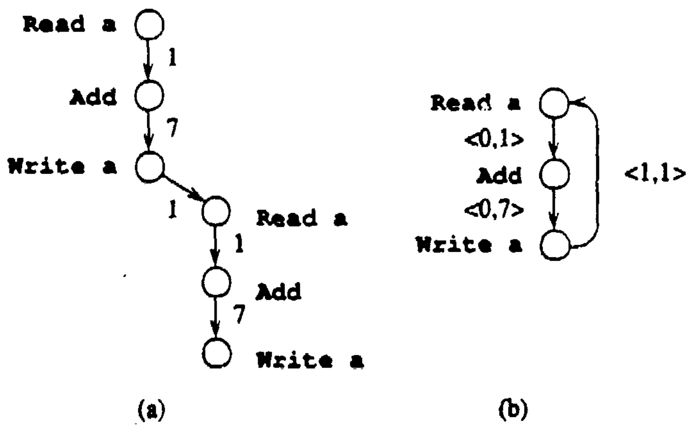

Figure 5-4: (a) Delays between operations from two iterations, and (b) precedence graph

meeting the lower bound is optimal. Empirical results show that this lower bound can indeed be met in many cases [18].

5.2. The Algorithm

The problem of finding the optimal software pipeline schedule is NP-complete. For acyclic graphs, the scheduling problem is tractable if operations execute in unit time and use only one resource. The polycyclic architecture [22] and the Cydra 5 architecture [9] use a specialized, rather expensive crossbar to provide exactly that property. All functional units of a polycyclic machine are interconnected through a crossbar. This crossbar has storage at every crosspoint to serve as a dedicated buffer for each pair of functional units. Therefore, there is never any contention in reading or writing data. Each operation thus consumes only one explicitly scheduled resource. For acyclic graphs, the minimum initiation interval is given by the bound discussed above and an optimal schedule can easily be found. However, the problem remains NP-complete for cyclic graphs even if operations use one unit of resource and execute in one unit time.

Without the specialized hardware to support software pipelining, both the FPS and the Warp/iWarp compilers use software heuristics. The algorithms used for scheduling acyclic graphs are similar, but the cyclic graph scheduling algorithm is significantly improved in our Warp/iWarp compilers. The algorithm for acyclic graphs is as follows: First, establish a lower and an upper bound on the initiation interval. The lower bound is calculated from the resource and precedence constraints; the upper bound can be found by the schedule of a single loop iteration. Next, find the smallest initiation interval. Simple linear search is used in our Warp/iWarp compiler because empirical results show that a schedule meeting the lower bound can often be found. The algorithm first sets the target of the initiation interval to be the lower bound value, and attempts to find a pipelinable schedule for the target initiation interval using the method described below. If the attempt fails, this process is reiterated by increasing the target initiation interval by one clock tick at a time.

The basic algorithm used to find a software pipelinable schedule for a target initiation is list scheduling. In list scheduling, the precedence constraints are applied first to determine the earliest slot in which an operation can be scheduled. The scheduler then goes on to try to satisfy the resource constraints; the modulo resource reservation table defined above is used to determine if there is a resource conflict. The scheduler tries to schedule the operation in successive time slots until one that can accommodate its resource requirement is found. If s is the target initiation interval, and s attempts to satisfy the resource constraints fail, by the definition of modulo resource usage, this operation cannot fit into the schedule built so far. When this happens, the attempt to find a schedule for the given initiation interval is aborted and the scheduling process is repeated with a greater interval value.

As in the case of acyclic graphs, the main scheduling step for cyclic graphs is iterative. For each target initiation interval, the strongly connected components are first scheduled individually. The original graph is then reduced by representing each strongly connected component as a single vertex: the resource usage of the vertex represents the aggregate resource usage of its components, and edges connecting nodes from different connected components are represented by edges between the corresponding vertices. This reduced graph is acyclic, and the acyclic graph scheduling algorithm can then be applied.

Two main concepts are used in the algorithm for scheduling the strongly connected components. First, the precedence constraints are preprocessed so that the scheduler can easily determine the legal time span in which any node can be scheduled. Second, the order in which the instructions are scheduled is designed such that when the target initiation interval value is increased, the chance for success also improves. This is important because it would be futile if the scheduling algorithm simply retried the same schedule that failed.

A large set of evaluation data on the Warp/iWarp machine indicates that provably optimal schedules can often be found [18]. This shows that software pipelining does not require expensive hardware support. The code generated is compact; the body of a software pipelined loop is even shorter than the unoptimized code.

6. Hierarchical Reduction

The *hierarchical reduction* technique is designed to make software pipelining applicable to all innermost loops, including those containing conditional statements. The proposed approach schedules the program hierarchically, starting with the innermost control constructs. As each construct is scheduled, the entire construct is reduced to a simple node representing all the scheduling constraints of its components with other constructs. This node can then be scheduled just like a simple node within the surrounding control construct. The scheduling process is complete when an entire program is reduced to a single node.

The use of the construct structure exploits high-level control dependence knowledge [11] to increase the opportunity for code motion. As an example of the kind of code motions achievable with this technique, consider the following program:

```
FOR i := 0 to n DO
BEGIN
      statement 1;
      IF c THEN statement 2 ELSE statement 3;
      statement 4;
END
```

Although statement 4 comes after the conditional statement, it is not control dependent upon the result of the condition `c`. Once the program decides to execute another iteration, it can execute statements 1 and 4 in any order that satisfies the data dependences. For example, an operation in statement 4 can be executed before the conditional statement. The hierarchical reduction algorithm first schedules the `THEN` and `ELSE` parts of the conditional statement, and represents the entire construct with a single node that inherits the union of the scheduling constraints for each of the two parts of the conditional statement. The entire construct is then scheduled with statements 1 and 4. Operations corresponding to statements 1 and 4 may be reordered, they may also execute in parallel with the `THEN` and `ELSE` components of the conditional statement. At code emission time, any code scheduled in parallel with the conditional statement is duplicated in both the `THEN` and `ELSE` parts.

This control dependence knowledge when combined with software pipelining can produce surprisingly efficient code. The loop termination test for the next iteration can be performed immediately after the decision to execute the current iteration. This test can move past all the conditional branches in the body of the loop. In this way, hierarchical reduction exposes many more parallel operations for scheduling.

Hierarchical reduction also minimizes the penalty of short vectors, or loops with small number of iterations. The prolog and epilog of a loop can be overlapped with scalar operations outside the loop; the epilog of a loop can be overlapped with the prolog of the next loop; lastly, software pipelining can be applied even to an outer loop. In summary, hierarchical reduction makes it possible to exploit parallelism in a much larger set of applications. It allows loops containing conditional statements to be software pipelined, and it finds parallelism within loop bodies that are too long to pipeline.

7. Modulo Variable Expansion

If traditional register assignment were performed before code scheduling, then the reuse of registers for different variables would significantly reduce the potential parallelism in the code. This is because the objective of register assignment is to use as few registers as possible. A register is recycled in the shortest amount of time, thus creating many more data dependences that need to be observed. Cooperation is therefore required between code scheduling and register assignment in a superscalar compiler. Proposed strategies include combining register assignment with scheduling [15], and postponing register assignment until after scheduling [18]. The latter approach simplifies the compiler design by separating scheduling and register assignment into two different phases. The drawback, however, is that there may not be enough registers and code needs to be inserted to spill values to memory.

There is one form of register reuse that can greatly inhibit parallelization, and that is the use of the same register for the same variable in different iterations of a loop. To illustrate this point, let us use the same example:

```
FOR i:= 0 TO n DO
    A[i] := A[i]+1.0;
```

For the sake of simplicity, here we assume that a floating-point addition takes only two clocks. The object code for one iteration, complete with register assignment, is as follows.

```
#   R1 preloaded with address of A
#  FR7 preloaded with 1.0

LD   FR1,(R1)
FADD FR1,FR7
nop
ST   FR1,(R1)
ADD  R1,R1,4
```

The register assignment prevents this vectorizable loop from executing in parallel. The register FR1 cannot be loaded with the next input until after its last use in the previous iteration. Similarly, the register R1 cannot be incremented until the last store operation is performed. Anti-dependences force the write operations to follow all the read operations of the old values; consequently, the computation must execute serially.

Modulo variable expansion is a register assignment technique that eliminates these anti-dependences. The following is the result of applying the combination of software pipelining and modulo variable expansion to the example above.

```
    #  R1 preloaded with address of A
    # FR7 preloaded with 1.0

    LD   FR1,(R1)
    FADD FR1,FR7   ADD  R2,R1,4
 1:                LD   FR2,(R2)
    ST   FR1,(R1)  FADD FR2,FR7   ADD  R1,R2,4
                                  LD   FR1,(R1)
                   ST   FR2,(R2)  FADD FR1,FR7   ADD  R2,R1,4  BLoop 1
                                  nop
                                  ST   FR1,(R1)
```

To eliminate the anti-dependence constraint, the second iteration uses a different set of registers, R2 and FR2, and can thus overlap with the first. The third iteration, on the other hand, can reuse the set in the first iteration. In fact, every other iteration can use the same set of registers, making the code identical every two consecutive iterations. The length of the steady state is just twice the initiation interval and the loop body is therefore still very compact.

We call this optimization of assigning several registers to a loop variable modulo variable expansion. In vectorizing compilers, scalar variables are expanded into arrays so that each iteration refers to a different array element, making the loop vectorizable. Modulo variable expansion takes advantage of the flexibility of superscalar machines, and reduces the number of registers allocated to a variable by reusing the same location in non-overlapping iterations.

A tradeoff can be made between the degree of loop unrolling and the number of registers used. For the Warp machine which contains a relatively large number of registers, minimizing the degree of unrolling is a better choice [19]. Eisenbeis et al., on the other hand, minimizes register usage because their target machine, Cray-2, has only eight vector registers [10].

8. Performance of Superscalar Machines

Having functional units that can be explicitly controlled by software, a superscalar processor is more versatile than a vector machine. The parallelism on a vector machine is restricted to the set of vector instructions, and, if chaining is supported, parallelism between vector instructions that use different functional units. Using software pipelining to schedule a superscalar with similar functional units, a simple loop that corresponds to a vector instruction, such as the pairwise additions of two vectors, can execute at the same throughput rate as a vector instruction. In addition, a superscalar can find parallelism in complex loops. Loops do not need to be decomposed into simple vector instructions which require partial expressions be buffered in vector registers. More importantly, a superscalar can find parallelism in loops with recurrences and conditional statements.

The ability of a superscalar machine to execute custom generated parallel code eliminates the need for buffering vectors of partial results. For example, a vectorizing compiler must decompose the loop in Figure 8-1(a) into two, each corresponding to a vector-add instruction (Figure 8-1 (b)). The partial sums must be buffered in a vector register. On a superscalar machine, the partial results can be operated on as soon as they are generated, as illustrated in Figure 8-1(c). This reduces the number of registers needed and possibly memory accesses.

```
(a)   FOR i := 0 TO n DO BEGIN
         c[i] := a[i]+b[i]+c[i];
      END;

(b)   FOR i := 0 TO n DO BEGIN
         t[i] := a[i]+b[i];
      END;
      FOR i := 0 TO n DO BEGIN
         c[i] := t[i]+c[i];
      END;

(c)   FOR i := 0 TO n DO BEGIN
         t := a[i]+b[i];
         c[i] := t+c[i];
      END;
```

Figure 8-1: Reduced register requirement in superscalar machines (a) source program, (b) vector code, (c) scalar code.

A recurrence does not necessarily mean serial execution for superscalar machines. As long as there are other operations that can execute in parallel with the recurrence computation, a high computation rate can still be obtained using software pipelining. The degree of parallelism in a vectorized loop is of the order of the number of iterations in the loop. A recurrence, however, limits the degree of parallelism by the ratio of independent operations to the length of the cyclic dependence. This limited form of parallelism can be exploited in superscalar processors because of their unique zero synchronization overhead. The compiler strategy for superscalar machines is different from that for vector machines. A vectorizing compiler tries to decompose a loop into smaller loops, separating recurrences from vectorizable loops. A superscalar compiler, on the other hand, tries to jam independent loops together. The vectorizable loop may be executed on the idle functional units while the program computes a recurrence!

In addition to recurrences, hierarchical reduction allows us to find parallelism even in loops with conditional statements. Hierarchical reduction also reduces the penalty typically associated with short vectors. In a superscalar machine, the scalar operations can be overlapped with the prolog and epilog of a software pipelined loop. This easy integration of scalar and vector operations makes the performance of the system less sensitive to the size of the data. Moreover, software pipelining can be applied even to outer loops, making the advantages of software pipelining applicable even for programs containing short innermost loops.

The instruction scheduling and register assignment techniques have been implemented in the compilers for the Warp and iWarp machines, and have been extensively evaluated [18]. The Warp processor has a peak computation rate of 10 MFLOPS, an impressive performance for a machine built in 1986. This peak computation rate is achieved by a high degree of parallelism and specialization. In a single instruction, a Warp processor can perform one 7-staged floating-point addition, one 7-staged floating-point multiplication, one memory read, one memory write, two integer operations, and one branch operation.

We have analyzed the performance of a set of seventy-two programs and the Livermore kernels. The performance of most of the programs fall between the 1 to 4 MFLOPS range, with a 2.8 MFLOPS average. This utilization of resources is higher than that typically observed in supercomputers. Performance analysis of the software pipeliner shows that the scheduler is successful in exploiting parallelism once the parallelism is detected. About three-quarters of over one hundred loops pipelined are provably optimal. When compared with code generated by a compiler that finds parallelism only within a basic block, most of the loops achieve a speed up of between two and six.

9. Conclusions

This paper presents an overview of compiler optimizations that exploit parallelism in a superscalar machine. High-level loop transformations improve data locality and place parallelism in the innermost loops, in preparation for instruction level optimizations. Software pipelining is the basic technique that finds parallelism across iterations in inner loops. Hierarchical reduction helps deliver a high level of performance for a broader range of applications, for example, by permitting software pipelining to be used even for loops with conditional statements. And lastly, modulo variable expansion eliminates dependence constraints due to reuse of registers between iterations.

The superscalar architecture is a promising alternative to vector machines. We now have compiler techniques that can generate highly efficient parallel code directly from user programs. Given the same hardware functional units, a superscalar machine delivers the same performance of a vector machine if the program is vectorizable. And the superscalar machine is decidably superior to vector machines when the computation contains recurrences and conditional statements. A superscalar does not exhibit a dichotomy in performance depending on whether the code is vectorizable or not.

Compiler optimizations require programs to be analyzable statically. A superscalar architecture has an organization that is more easily enhanced to handle programs that are not amenable to static analysis. By a judicious use of hardware to provide dynamic information to cooperating software, processors that deliver a consistent high-performance through instruction level parallelism are possible.

Acknowledgments

The research on instruction scheduling and register assignment was performed as part of the Warp/iWarp systolic array projects at Carnegie Mellon University. I'd like to thank H. T. Kung, my thesis advisor, for the guidance in this research, and Thomas Gross for his indispensable contribution to the compiler design and construction, and the rest of the Warp/iWarp team for their contribution to the projects. The work on data locality was performed here at Stanford with Michael Wolf. The research is supported in part by DARPA contract N00014-87-K-0828.

References

[1] Allen, R. and Kennedy, K.
Automatic Translation of FORTRAN Programs to Vector Form.
ACM Transactions on Programming Languages and Systems 9(4):491-542, October, 1987.

[2] Anderson, D. W., Sparacio, F. J., and Tomasulo, R. M.
The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling.
IBM Journal of Research and Development :8-24, Jan., 1967.

[3] Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O. and Webb, J. A.
The Warp Computer: Architecture, Implementation and Performance.
IEEE Trans. on Computers C-36(12):1523-1538, Dec., 1987.

[4] Banerjee, U.
Dependence Analysis for Supercomputing.
Kluwer Academic Publishers, 1988.

[5] Bloch, E.
The Engineering Design of the Stretch Computer.
In *Proc. Eastern Joint Computer Conference*, pages 48-58. 1959.

[6] Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H. T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P. S., Sutton, J., Urbanski, J. and Webb, J.
iWarp: An Integrated Solution to High-Speed Parallel Computing.
In *Proc. Supercomputing '88*, pages 330-339. Nov., 1988.

[7] Cohn, R., Gross, T., Lam, M. and Tseng, P. S.
Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor.
In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 2-14. April, 1989.

[8] Colwell, R. P., Nix, R. P., O'Donnell, J. J., Papworth, D. B., and Rodman, P. K.
A VLIW Architecture for a Trace Scheduling Compiler.
IEEE Trans. on Computers C-37(8):967-979, Aug., 1988.

[9] Dehnert, J. C., Hsu, P. Y.-T. and Bratt, J. P.
Overlapped Loop Support in the Cydra 5.
In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 26-38. April, 1989.

[10] Eisenbeis, C., Jalby, W. and Lichnewsky, A.
Squeezing More CPU Performance out of a Cray-2 by Vector Block Scheduling.
In *Proc. Supercomputing 88*. 1988.

[11] Ferrante, J., Ottenstein, K. and Warren, J.
The Program Dependence Graph and its Use in Optimization.
ACM Trans. on Programming Languages and Systems, July, 1987.

[12] Fisher, J. A.
Trace Scheduling: A Technique for Global Microcode Compaction.
IEEE Transactions on Computers C-30(7):478-490, July, 1981.

[13] Fisher, J. A.
The VLIW Machine: A Multiprocessor for Compiling Scientific Code.
Computer :45-53, July, 1984.

[14] Gannon, D., Jalby, W., and Gallivan, K.
Strategies for Cache and Local Memory Management by Global Program Transformation.
Journal of Parallel and Distributed Computing 5:587-616, 1988.

[15] Goodman, J. R. and Hsu, W. C.
Code Scheduling and Register Allocation in Large Basic Blocks.
In *Proc. 1988 International Conference on Supercomputing*, pages 442-452. July, 1988.

[16] Gross, T. and Lam, M.
Compilation for a High-Performance Systolic Array.
In *Proc. ACM SIGPLAN 86 Symposium on Compiler Construction*, pages 27-38. June, 1986.

[17] Kohn, L. and Fu, S.-W.
A 1,000,000 Transistor Microprocessor.
In *Proc. 1989 International Solid-State Circuits Conference Digest of Technical Papers*, pages 54-55. Feb., 1989.

[18] Lam, M.
A Systolic Array Optimizing Compiler.
PhD thesis, Carnegie Mellon University, May, 1987.

[19] Lam, M.
Software Pipelining: An Effective Scheduling Technique for VLIW Machines.
In *ACM Sigplan '88 Conference on Programming Language Design and Implementation*.. June, 1988.

[20] Patel, J. H. and Davidson, E. S.
Improving the Throughput of a Pipeline by Insertion of Delays.
In *Proc. 3rd Annual Symposium on Computer Architecture*, pages 159-164. Jan., 1976.

[21] Porterfield, A.
Compiler Management of Program Locality.
Technical Report, Rice University, Jan, 1988.

[22] Rau, B. R. and Glaeser, C. D.
Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing.
In *Proc. 14th Annual Workshop on Microprogramming*, pages 183-198. Oct., 1981.

[23] Rau, B. R., Kuekes, P. J. and Glaeser, C. D.
A Statically Scheduled VLSI Interconnect for Parallel Processors.
In *VLSI Systems and Computations*, pages 389-395. October, 1981.

[24] Thornton, J. E.
Parallel Operation in the Control Data 6600.
In *AFIPS Conference Proceedings FJCC*, pages 33-40. 1964.

[25] Touzeau, R. F.
A Fortran Compiler for the FPS-164 Scientific Computer.
In *Proc. ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 48-57. June, 1984.

[26] Wolf, M. E. and Lam, M. S.
An Algorithm to Generate Sequential and Parallel Code with Improved Data Locality.
1989.

[27] Wolfe, M. J.
Optimizing Supercompilers for Supercomputers.
PhD thesis, University of Illinois at Urbana-Champaign, October, 1982.